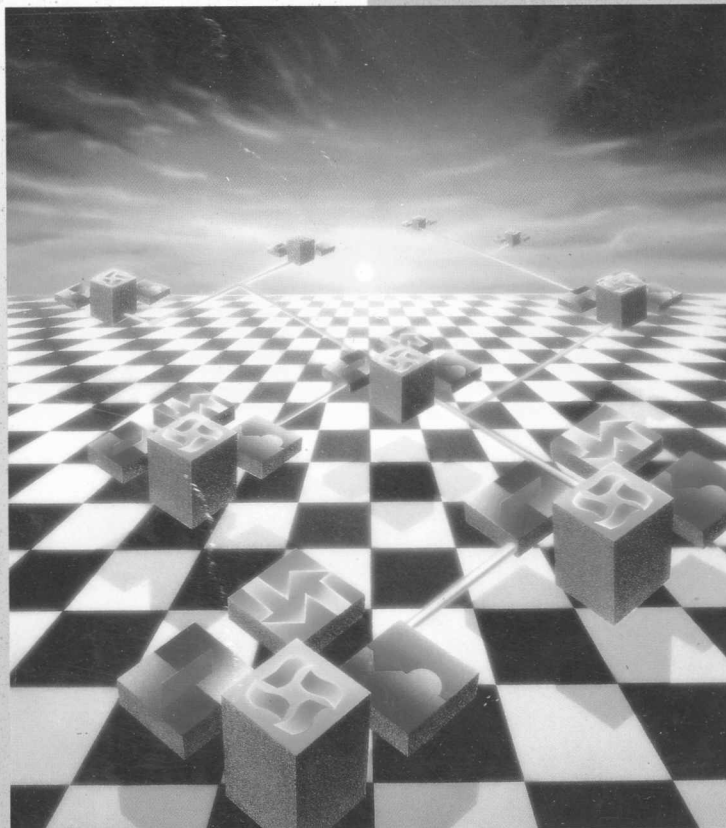
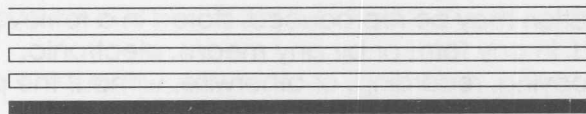


AK

NEURON[®] C PROGRAMMER'S GUIDE



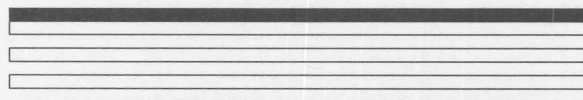
 ECHELON



NEURON[®] C Programmer's Guide

Revision 2

ECHOLON[®]
Corporation, Inc.



No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Echelon Corporation.

Echelon, LON, and NEURON are registered trademarks of Echelon Corporation. LONBUILDER, LONMANAGER, LONTALK, and LONWORKS are trademarks of Echelon Corporation.

Document No. 29300

Printed in the United States of America.
Copyright ©1990, 1991, 1992 by Echelon Corporation

Echelon Corporation
4015 Miranda Avenue
Palo Alto, California
94304

Contents

NEURON C Programmer's Guide

Preface	vii
Audience	viii
Content	viii
Related Manuals	ix
How to Use this Manual	ix
Typographic Conventions for Syntax	xi
Chapter 1 Overview	1-1
What Is NEURON C?	1-2
What is a LONWORKS Application?	1-3
Network Management Tools	1-5
Standard Network Variable Types (SNVTs)	1-7
Designing a LONWORKS Application	1-9
Differences between NEURON C and ANSI C	1-20
NEURON C Variable Types	1-22
NEURON C Storage Classes	1-22
Variable Initialization	1-24
Integer Constant Typing	1-25
NEURON C Declarations	1-26
Compiler Directives	1-27

Chapter 2 Focusing on a Single Node	2-1
What Happens on a Single Node?	2-2
The Scheduler	2-2
Timers	2-12
Input/Output	2-16
Examples	2-52
Input Clock Frequency and Timer Accuracy	2-60
Delay Functions	2-66
Chapter 3 How Nodes Communicate Using Network Variables	3-1
Major Topics	3-2
Overview	3-3
Declaring Network Variables	3-7
Connecting Network Variables	3-16
Network Variable Events	3-24
Synchronous Network Variables	3-30
Processing Completion Events for Network Variables	3-33
Polling Network Variables	3-34
Monitoring Network Variables	3-39
Authentication	3-41
Chapter 4 How Nodes Communicate Using Explicit Messages	4-1
Explicit Messages vs. Network Variables	4-2
Layers of Neuron Software	4-3
Implicit Messages: Network Variables	4-4
Explicit Messages	4-5
Constructing a Message	4-6
Sending a Message	4-14
Receiving a Message	4-15
Example	4-19
Explicit Addressing	4-21
Sending a Message with the ACKD Service	4-22
Processing Completion Events for Messages	4-24
Preemption Mode and Messages	4-26
Asynchronous and Direct Event Processing	4-28
Using the Request/Response Mechanism	4-29
Buffers	4-36

Chapter 5 Additional Features	5-1
The Scheduler	5-2
Scheduler Reset Mechanism	5-4
Bypass Mode	5-6
Watchdog Timer	5-7
Additional Predefined Events	5-8
Sleep Mode	5-10
Error Handling	5-15
Access to Node Error Status	5-17
Chapter 6 Memory Management	6-1
Reallocating On-Chip EEPROM	6-2
Allocating Buffers	6-3
Using NEURON CHIP Memory	6-14
Memory Use	6-22
Usage Tip for Memory Mapped I/O	6-23
What to Try When a Program Doesn't Fit on a NEURON 3120 CHIP	6-24
System Library on a NEURON 3120 CHIP	6-31
Appendix A Sample Application	A-1
Introduction	A-2
Door Node	A-4
Interior Light Node	A-8
Key Node	A-13
Headlight Switch Node	A-15
Headlight Node	A-18
Appendix B Syntax Summary	B-1
Syntax Conventions	B-2
NEURON C External Definitions	B-2
Enumeration Syntax	B-11
Structure/Union Syntax	B-12
Declarator Syntax	B-13
Conditional Events	B-16
Statements	B-18
ANSI C Expressions	B-19
Built-in Variables and Functions	B-22
Limits.h	B-24

Appendix C NEURON C Language Reference	C-1
C.1 Predefined Events	C-2
C.2 Functions	C-22
C.3 Network Variable Declarations	C-76
C.4 Timer Declarations	C-82
C.5 Built-in Variables and Objects	C-83
C.6 Reserved Words	C-90
C.7 I/O Objects	C-95
Appendix D Implementation Dependencies	D-1
Appendix E Compiler Error Messages	E-1
Introduction	E-2
Compiler Error Messages	E-3
Appendix F System Error Messages	F-1
Index	I-1
 Figures	
Figure 1-1. Identifying the Nodes	1-11
Figure 1-2. Multiple I/O Devices Connected to a Single Node	1-11
Figure 1-3. External Interface Definitions	1-13
Figure 1-4. Two Nodes Using the Same Network Variable	1-15
Figure 1-5. Sample Development Network with Five Nodes	1-19
Figure 1-6. Diagram of Standard Program Identification Fields	1-35
Figure 2-1. Flow Diagram for Timer/Counter Circuits	2-36
Figure 2-2. Possible Master/Slave Connections for the NEURON CHIP	2-43
Figure 2-3. Pin Assignments for the Three Modes of Parallel I/O	2-44
Figure 2-4. Handshake Protocol Sequence Between Master and Slave	2-46
Figure 2-5. Transferring Data from Master to Slave A	2-50
Figure 2-6. Transferring Data from Microprocessor to Slave B	2-51
Figure 2-7. Sample Thermostat Node	2-53
Figure 2-8. Sample Dimmer Node	2-57
Figure 2-9. Neurowire Connection to a Display	2-59
Figure 2-10. Expected, Low, and High Duration of Timeout Events	2-64

Figures (cont)

Figure 3-1.	Sample Development Network with Five Nodes	3-4
Figure 3-2.	Sample Automobile Application	3-18
Figure 3-3.	Nodes in the Sample Automobile Application	3-20
Figure 3-4.	Network Variables for Doors and Interior Light	3-21
Figure 3-5.	Network Variables for Headlight Switch, Key Switch, and Headlights	3-22
Figure 3-6.	Expanded Automobile Application	3-23
Figure 3-7.	Authentication Process	3-44
Figure 4-1.	Sending a Message	4-3
Figure 5-1	NEURON CHIP Firmware Scheduling of Nonpriority and Priority When Clauses	5-3
Figure 6-1.	Application and Network Buffers	6-3
Figure 6-2.	System and Protocol Overhead in Application and Network Buffers	6-4
Figure 6-3.	Off-Chip Memory on the NEURON 3150 CHIP	6-14
Figure 6-4.	Memory Maps for NEURON 3150 CHIP and NEURON 3120 CHIP, Showing Areas Defined by Linker	6-15
Figure A-1.	Sample Auto Control System	A-3
Figure C-1.	Bitshift Output	C-100
Figure C-2.	Quadrature Input	C-127
Figure C-3.	Triac Output, Example 1 (NEURON 3120 CHIP Only)	C-135
Figure C-4.	Triac Output, Example 2 (NEURON 3120 CHIP Only)	C-136
Figure C-5.	Triggeredcount Output Object	C-138

Tables

Table 1-1.	Standard Program Identification Fields	1-34
Table 2-1.	I/O Object Types	2-19
Table 2-2.	I/O Devices	2-24
Table 4-1.	Ranges for Message Codes	4-11
Table 4-2.	Success/Failure Completion Events	4-24
Table 6-1.	Values for Buffer Sizes and Counts	6-11
---	Events Listed by Functional Group	C-2
---	Functions Listed by Functional Group	C-24

■ General Changes

- Change references to the *NEURON CHIP Advanced Information* document to *NEURON 3150 and 3120 CHIP Data Book* throughout your manual.
- Change references to *LONWORKS Network Interface Development Guide* to *LONBUILDER Microprocessor Interface Program (MIP) User's Guide* throughout your manual.

Preface

This manual describes how to write programs using NEURON® C. NEURON C is a programming language based on ANSI C, with extensions to support the development of distributed control networks. Key concepts in programming with NEURON C are explained through the use of specific code examples and diagrams. A general methodology for designing and implementing a distributed application is also presented.

Audience

The *NEURON C Programmer's Guide* is intended for application programmers who are developing LONWORKS™ applications. Readers of this guide are assumed to have some C programming experience.

For a complete description of ANSI C consult the following references:

- American National Standard X3.159-1989, Programming Language C, D.F. Prosser, American National Standards Institute, 1989.
- *Standard C: Programmer's Quick Reference*, P. J. Plauger and Jim Brodie, Microsoft Press, 1989.
- *C: A Reference Manual*, Samuel P. Harbison and Guy L. Steele, Jr., 3rd edition, Prentice-Hall, Inc., 1991.
- *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, 2nd edition, Prentice-Hall, Inc., 1988.

Content

The *NEURON C Programmer's Guide*:

- Outlines a recommended general approach to developing a LONWORKS application.
- Explains key concepts of programming in NEURON C through the use of code fragments and examples.
- Provides a complete reference section on NEURON C (Appendix C).

Related Manuals

The *LONBUILDER™ User's Guide* (part no. 078-0002-01A) lists and describes all tasks related to LONWORKS application development using the LONBUILDER™ Developer's Workbench. Refer to that guide for detailed information on the user interface to the LONBUILDER Developer's Workbench.

How to Use this Manual

Chapters 1 through 4 present the basics of writing NEURON C programs. If your time is limited, be sure to read those chapters first. Additional topics are covered in Chapters 5 and 6.

The *NEURON C Programmer's Guide* is organized into six chapters. These chapters have the following content:

Chapter 1, *Overview*, describes a methodology for designing and implementing a distributed application. It compares NEURON C to ANSI C, describes implementation dependencies for NEURON C, and describes NEURON C data types, storage classes, and compiler directives.

Chapters 2 through 4 cover the specifics of how to write a NEURON C program. For purposes of discussion, Chapter 2 focuses on topics relating mostly to individual nodes, and Chapters 3 and 4 explore the topics of network variables and explicit messages, which provide the interface between nodes.

Chapter 2, *What Happens on a Single Node?*, describes the NEURON CHIP event scheduling mechanism, as well as the declaration and use of timer objects and I/O objects.

Chapter 3, *How Nodes Communicate Using Network Variables*, describes how to declare and use network variables.

Chapter 4, *How Nodes Communicate Using Messages*, describes how to declare, send, and receive explicit messages.

Chapter 5, *Additional Features*, includes discussion of the scheduler reset mechanism, bypass mode, sleep mode, additional predefined events, and error handling.

Chapter 6, *Memory Management*, describes how to reallocate on-chip EEPROM, how to allocate buffers, and how to use off-chip memory.

The appendices provide additional material, as follows:

Appendix A contains the object diagram and programs for the sample application discussed in Chapter 3.

Appendix B contains the syntax summary for NEURON C.

Appendix C is the NEURON C Language Reference. It contains the following sections:

- C.1 Predefined Events
- C.2 Functions
- C.3 Network Variable Declarations
- C.4 Timer Declarations
- C.5 Built-in Variables and Objects
- C.6 Reserved Words
- C.7 I/O Devices

Appendix D discusses implementation dependencies.

Appendix E lists NEURON C compiler error messages and offers suggestions on how to identify and correct problems.

Appendix F lists and describes the system error messages for the NEURON CHIP firmware.

Typographic Conventions for Syntax

This manual uses the following typographic conventions for syntax:

<i>Type</i>	<i>Used For</i>	<i>Example</i>
boldface type	keywords	network
<i>Italic type</i>	user-defined names or keyword choices	<i>identifier</i>
square brackets	optional fields	[<i>connection-info</i>]
vertical bar	a choice between two elements	input output

For example, the syntax for declaring a network variable is:

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [= *initial-value*];

Punctuation other than square brackets and vertical bars is required where used (parentheses, semicolons).

Code fragments and examples appear in the Courier font:

```
ram
eeprom
signed
int
switch_on
```


1

Overview

This chapter is divided into two parts. The first part introduces you to programming with NEURON C. It describes the basic steps involved in designing and implementing a distributed application using LONWORKS™ technology. The introduction uses a simple example to describe the general methodology for developing both small and complex LONWORKS applications and to introduce key concepts in NEURON C such as nodes and network variables.

The second part of this chapter includes important introductory material on NEURON C concerning NEURON C types, storage classes, compiler directives, and how NEURON C compares to ANSI C.

What Is NEURON C?

NEURON C is a programming language designed for NEURON CHIPS and based on ANSI C. It includes extensions to ANSI C that directly support the NEURON CHIP firmware, which make it a powerful tool for the development of LONWORKS applications. Some of these features are:

- a new class of objects, *network variables*, which simplify data sharing among nodes
- a new statement type, the *when statement*, which introduces *events* and defines the temporal ordering of these events
- explicit control of I/O operations, through declaration of *I/O objects*, to standardize multi-functional I/O specific to NEURON CHIPS
- support for explicit message passing, used for direct access to the underlying LONTALK™ protocol services

NEURON C provides a special set of objects for the distributed LONWORKS environment as well as a set of built-in functions for accessing these objects. Experienced C programmers will find NEURON C a natural extension to the familiar C paradigm. NEURON C offers built-in type checking and allows the programmer to generate highly efficient code for distributed LONWORKS applications.

NEURON C omits ANSI C features not required by the standard for free-standing implementations. For example, certain standard C libraries are not part of NEURON C. (See *Differences between NEURON C and ANSI C* later in this chapter.)

What is a LONWORKS Application?

LONWORKS applications consist of intelligent devices, called nodes, that communicate over a control network using the LONTALK protocol.

Individual nodes contain application code that deals with objects tailored to the LONWORKS environment, including *network variables*, which provide a well-defined interface between the nodes in a LONWORKS application and *Input/Output (I/O) objects* which provide a common interface to application I/O devices.

The LONTALK™ protocol includes a predefined set of *Standard Network Variable Types (SNVTs)* that enhance compatibility between network variables on different nodes. SNVTs also have properties that make node installation easier and that promote interoperability among products.

Nodes

Nodes are objects that interact with physically attached I/O devices and communicate with other nodes over a network, using the LONTALK protocol. All nodes include a NEURON CHIP for communication and control, an I/O interface for one or more I/O devices, and a transceiver to connect to the network. The node's behavior is defined by a program and configuration information contained in the node's memory. The user-definable portion of node memory consists of two parts: the application image and the network image.

The *application image* includes the node's application program, which defines the events to which the node responds and the actions it takes in response to those events. Part of a node's memory contains this code. This manual discusses how to write a node's application program in NEURON C.

The *network image* defines the relationship of a node to other nodes in the network and gives it a unique location in the network. The network image is created using commands from a special type of node called a network management tool. The *LONBUILDER User's Guide* describes the process of customizing a node using the network management tool that is built into the LONBUILDER Developer's Workbench. Network management tools are generally not required for LONWORKS networks to operate. However, they are useful in facilitating installation and maintenance of nodes and networks.

In general, products that use LONWORKS nodes will receive their application image during manufacture. The nodes will typically receive their network images in the field, when the product is installed in a specific network. However, it is possible to build nodes that can receive their application images in the field or their network images during manufacture. LONWORKS offers great flexibility; it allows many options for installing and configuring nodes.

For the majority of nodes, the NEURON CHIP executes the node's application program and implements the LONTALK protocol. In some nodes, such as those that are used for network management or monitoring, the node application program may execute on a non-NEURON CHIP host, which could be a PC-compatible system, a workstation, a minicomputer, or another microprocessor. These nodes still contain a NEURON CHIP; however, the NEURON CHIP in this case is used solely as a LONTALK protocol processor. This manual discusses application programming for NEURON CHIP-hosted nodes only. The *Host Application Programmer's Guide* describes the development of nodes that include a host processor in addition to a NEURON CHIP.

Network Management Tools

Network management tools for simple networks can be built using NEURON C. Network management tools for more complex networks are built with the LONMANAGER API. See the *NEURON CHIP-based Installation of LONWORKS Networks* Engineering Bulletin (part no. 005-0022-01) for more information on NEURON CHIP-based installation; see the *LONMANAGER API Programmer's Guide* (model no. 39100) for more information on the development of network management tools using the LONMANAGER API.

Network management tools use a definition of the nodes' application interfaces to generate network configuration information and to install nodes on a network. The interface definitions may either be read from the nodes if SNVTs are used, or may be read from an external interface file. The external interface file is generated by the NEURON C compiler and provides a complete definition of a node's network interface. The external interface file contains the node's network variable names and types, information about messages, bit rate, and other information needed to install the node on a network.

During installation, nodes can be identified by the network management tool in three different ways:

- Grounding the service pin on a NEURON CHIP tells the NEURON CHIP firmware on that chip to send its NEURON ID out over the network. The network management tool uses the ID to download network and application parameters over the network.

- The nodes are installed unconfigured (no network image). The network management tool uses a network management message to get the NEURON IDs from every unconfigured node. Then the network management tool sends a wink command to each unconfigured node, one at a time. Nodes which receive this message and have a software task to handle the wink message identify themselves. For example, if the node controls a light, it could turn the light on and off, identifying its physical location. An installer identifies the location of the winking node to the network management tool.
- The installer can manually enter the NEURON ID, either through keyboard entry or a bar-code scanner.

A customized network management tool has a user interface tailored to a particular application. For example, a heating/cooling/lighting/security system for a commercial building might have a graphic user interface, which requests information on building functions and systems, to simplify the process of specifying node locations and how nodes interconnect. A user interface for a factory might present a different view of this network and its functions.

I/O Devices

A NEURON CHIP may be connected to one or more physical I/O devices. Examples of simple I/O devices include temperature and position sensors, valves, switches, and LED displays. NEURON CHIPS can also be connected to other microprocessors. I/O devices are discussed in detail in Chapter 2 and Appendix C.

Network Variables

A network variable is an object on one node that can be connected to network variables on one or more additional nodes. A node's network variables define its inputs and outputs from a network point of view and allow the sharing of data in a distributed application. Whenever a program writes into one of its *output* network variables, the new value of the network variable is propagated across the network to

all nodes with *input* network variables connected to that output network variable. Although the propagation of network variables occurs through LONTALK messages, these messages are sent “automatically”. The application program does not need any explicit instructions for sending and receiving network variable updates.

Network variables greatly simplify the process of developing and installing distributed systems because nodes can be defined individually, then connected and reconnected easily into new LONWORKS applications. Network variables are discussed in detail in Chapter 3 and Appendix C.

Standard Network Variable Types (SNVTs)

Network variables promote interoperability between nodes by providing a well-defined interface that nodes use to communicate. Interoperability simplifies installation of nodes into different types of networks by keeping the network configuration independent of the node's application. A node may be installed in a network and logically connected to other nodes in the network as long as the data types (for example, `int` or `long`) match. To further promote interoperability, the LONTALK protocol provides Standard Network Variable Types (SNVTs). SNVTs are a set of predefined types with associated units, such as degrees C, volts, or meters.

A network management tool can use LONTALK network management commands to automatically determine the type of every network variable declared as a SNVT.

SNVTs also provide a “Self-Documentation” (SD) feature. The application programmer can use this feature to create a text string including a network variable name, special installation instructions, etc. This information is stored with the application program on the node.

A node usually contains information about itself and its network variables. This information is called “Self-Identification” (SI).

NOTE: The node will contain SI information unless the compiler directive `#pragma disable_snvt_si` is used.

The `#pragma enable_sd_nv_names` may be used to include the network variable names in the Self-Documentation when SI is generated. In the SI information, each network variable in the node has associated with it up to four optional pieces of information: the network variable's name, a non-zero rate estimate in tenths of msgs/sec., a non-zero max rate estimate in tenths of msgs/sec., and an SD string for the network variable. The rate estimate and max rate estimate values are generated for any variable where specified in that variable's `bind_info`. If there is no value set in the `bind_info` for either rate estimate, the value defaults to zero and the value is omitted from the SD information.

Data-Driven vs. Command-Driven Protocols

Network variables are typically used to communicate data and state information between nodes. In command-based messaging systems, designers are faced with having a large number of commands specific to each application that must be managed, updated, and maintained. Each node has to have knowledge of every command. This leads to ever-growing command tables and application code. With network variables, the command or action portion of a message is not in the message. Instead, it is in the application program.

Event-Driven vs. Polled Scheduling

Although the example in this chapter uses an event-driven model for a distributed application, NEURON C allows you to construct polled applications that implement a centralized control model. Chapter 3 provides further information on polling.

In addition, NEURON C provides complete support for use of explicit messages, which can be used in place of the network variable approach described in this chapter, or to supplement this approach. Explicit messages are described in Chapter 4.

Interoperability Guidelines

Nodes designed in accordance with the LONWORKS interoperability guidelines can apply to carry the LONMARK™ logo. All LONMARK nodes will receive a standard program ID. This ID (stored in EEPROM) has space for the manufacturer ID, device class, device subclass, and model number. (See the *Compiler Directives* section at the end of this chapter.) Standard program IDs facilitate installation of nodes from multiple vendors, because a network management tool can use the ID to access a database containing further information about LONMARK nodes.

Designing a LONWORKS Application

A good network design requires an overall plan and a basic analysis of the parts of the network and how they interconnect. Although a LONWORKS application may ultimately perform complex and sophisticated tasks, the developer should first break down the problem into its simplest parts. Once this analysis is complete, the actual process of programming individual nodes becomes a relatively simple translation of object and task definitions into code.

Basic Steps in Designing the Application

The basic steps in designing a distributed application are listed here. Each step is described in more detail, with examples, in the following sections.

- 1 Define the problem.
- 2 Identify nodes and assign their functions.
- 3 Define the external interfaces for each node.
- 4 Write application programs for each node.
- 5 Debug and test individual nodes, one at a time.
- 6 Install nodes in a development network and test.

Step 1: Define the Problem

Although the design process requires you to focus on specific objects and tasks, it is usually helpful to define the problem in general terms first. Depending on the type of application, this step may be simple or complex. One example of a complex system might be a LONWORKS application for controlling a large number of functions in a high-rise office building, including lighting, heating and cooling, humidity, and security. Another sophisticated application might be an automated factory, in which hundreds of sensors and actuators are linked together to perform steps in a manufacturing process.

For this initial discussion, a simple example will suffice. Suppose you want to design a LONWORKS application in which a set of switches is used to control a set of lamps. You want to be able to connect and reconnect the lamps and switches so that any switch can control one or more lamps at the same time.

For purposes of this discussion, the functionality of this example is kept to a minimum. A complete LONWORKS lighting application would also probably include dimmers along with the switches to control lighting level. An occupancy sensor for each work area could also be connected to turn lights on and off automatically depending on whether the room was occupied.

Step 2: Identify Nodes and Assign Their Functions

This example has switch nodes and lamp nodes. The switches are in different physical locations in the room, so they need to be on separate nodes. Since you want individual control of the lamps, the lamps need to be on separate nodes (see Figure 1-1).

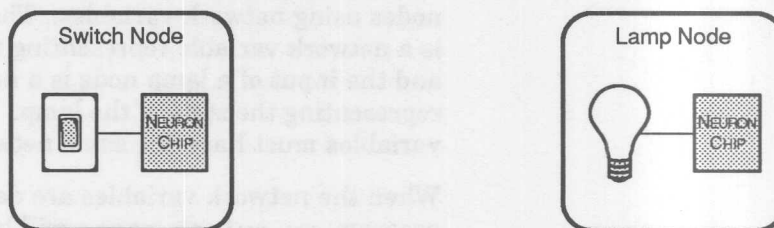


Figure 1-1. Identifying the Nodes

Although there is a one-to-one correspondence between nodes and I/O devices in this example, this is not a requirement. A number of I/O devices could be connected to a single node, as shown in Figure 1-2.

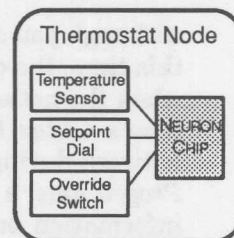


Figure 1-2. Multiple I/O Devices Connected to a Single Node

Step 3: Define the External Interfaces for Each Node

The next step is to define the external interfaces for each node. External interfaces are usually defined by network variables (described in Chapter 3) but may be defined by explicit messages (described in Chapter 4).

In this example, switch nodes will be connected to lamp nodes using network variables. The output of a switch node is a network variable representing the state of the switch; and the input of a lamp node is a network variable representing the state of the lamp. These two network variables must have the same network variable type.

When the network variables are declared in the switch program, `nv_switch_state` will be declared as an *output network variable* (that is, its node writes values that are propagated across the network). In the lamp program, the variable `nv_lamp_state` will be declared as an *input network variable* (that is, this node's network variable is updated from the network). Use of SNVTs simplifies the process of connecting nodes at a later time. See the *Standard Network Variable Types* section earlier in this chapter for more details.

Although you *define* the external interfaces for each node at this time, the connections are *implemented* as a separate step when the network is configured. See the *LONBUILDER User's Guide* for more information on configuring a network during development. See the *LONMANAGER API Programmer's Guide* (model no. 39100) for more information on configuring a network in the field. Changing connections between nodes does not require any modification of the application program, and the network can be reconfigured at any time in the future.

The external interface design for this sample application is shown in Figure 1-3.

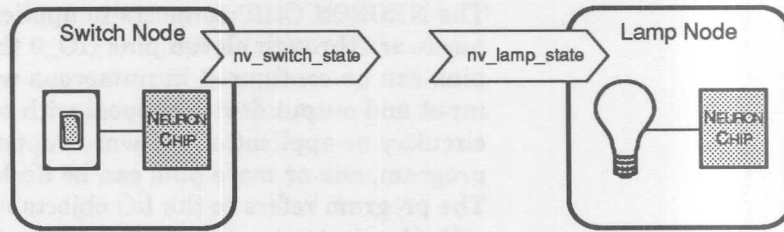


Figure 1-3. External Interface Definitions

Step 4: Write the Application Program for Each Node


NEURON C application programs are written by matching the functions of each node for NEURON C objects, defining tasks, and writing code to implement the objects and tasks. In this example, you can define the following functions for each node, with corresponding NEURON C objects for each function:

<i>Function for Switch Node</i>	<i>NEURON C Object</i>
Monitor a switch state	bit input I/O object
Send ON/OFF state	network variable output

<i>Function for Lamp Node</i>	<i>NEURON C Object</i>
Receive ON/OFF state	network variable input
Control lamp power	bit output I/O object

Additional functions might require other NEURON C objects, such as timers (see Chapter 2) or messages (see Chapter 4).

I/O Objects



The NEURON CHIP connects to application-specific hardware through eleven pins (IO_0 through IO_10). These pins can be configured in numerous ways to provide flexible input and output device support with a minimum of external circuitry or application software support. Within a program, one or more pins can be declared as I/O objects. The program refers to the I/O objects in NEURON C function calls (`io_in()`, `io_out()`) to perform the actual input/output operations during program execution.

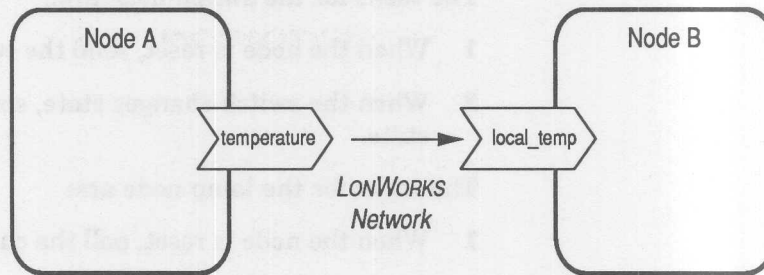
Network Variables

A network variable is an object shared by multiple nodes. For a given network variable, some nodes are *readers* of the variable and other nodes are *writers* that assign values to the variable (see Figure 1-4). Readers of a network variable declare the variable as a “network input”; writers declare a network variable as a “network output”.

An input network variable can be connected to an output network variable on the same node, so that a single node may be both a reader and a writer.

Several features distinguish network variables from regular C variables. First, just as two people might call the same person by different names, two nodes can have different names for the same network variable. In Figure 1-4, for example, Node A might name the network variable “temperature,” while Node B might name it “local_temp”. (The network management tool creates the connections between output network variables and input network variables.) In addition, an output network variable has the property that whenever its node assigns a new value to it, the new value is propagated over the network to all nodes with input network variables connected to that output network variable.

Node A writes values into an output network variable that it calls *temperature*.



Node B reads values from an input network variable that it calls *local_temp*.

Figure 1-4. Two Nodes Using the Same Network Variable

Network variables for each node are defined independently. Each switch node has a network variable that will be used to communicate the state of the switch. The network variables for the switches are named *nv_switch_state*. The network variable *nv_switch_state* will be used as *output* from a switch node to communicate its state.

Similarly, each lamp node has a network variable that will control the state of the lamp. In this example, the network variable for the lamp is named *nv_lamp_state*. Ultimately, the network variable *nv_lamp_state* will serve as *input* to a lamp node to tell it when to turn on and off.

Programming in NEURON C allows the designer to separate many program considerations from network considerations. At the program level, all of the switch nodes contain the same executable application with a network variable named *nv_switch_state*. This program can be reused in many different types of networks and on many different nodes. This is because connections are established when nodes are installed, rather than when they are programmed (see Step 6 later in this chapter).

Tasks

The tasks for the switch node are:

- 1 When the node is reset, send the current switch state.
- 2 When the switch changes state, send the new switch state.

The tasks for the lamp node are:

- 1 When the node is reset, poll the current lamp state.
- 2 When a new lamp state is received from the network, control the application I/O hardware to turn the light ON or OFF.

The NEURON CHIP firmware includes a built-in scheduler that handles the temporal ordering and execution of user-written tasks. The scheduling of tasks is *event-driven*. When a specified event becomes TRUE, the task associated with that event is executed.

Tasks can be assigned *priority*, so that their associated event is evaluated more frequently than nonpriority events.

Step 5: Debug and Test Individual Nodes

The next step is to debug and test each node, one at a time. The NV Browser feature in the LONBUILDER can be used to examine or change network variables.

The code for the switch/light application is listed below.

The following program is used in each of the lamp nodes:

```
// lamp.nc - Generic program for a lamp.  An input network
// variable controls the lamp's state.

#include <snvt_lev.h>
// I/O object declarations
IO_0 output bit io_lamp_out=0;
        // Turns an LED on or off, with no
        // external interface circuitry.

// Network variable declarations
network input SNVT_lev_disc nv_lamp_state;

// Event-driven code
when (reset)
{
    poll(nv_lamp_state);
}
when (nv_update_occurs(nv_lamp_state))
{
    // Use the network variable's value as the
    // new state for the lamp

    io_out(io_lamp_out, (nv_lamp_state!=ST_OFF)?1:0);
}
```

This program is also used in each of the switch nodes:

```
// switch.nc - Generic program for a switch. Set an output
// network variable when the switch changes state

#include <snvt_lev.h>
// I/O object declarations
IO_4 input bit io_switch_in;

// Network variable declarations
network output SNVT_lev_disc nv_switch_state = ST_OFF;

// Event-driven code
when (reset)
{
    nv_switch_state = io_in(io_switch_in) ? ST_ON : ST_OFF;
    io_change_init(io_switch_in);
}

when (io_changes(io_switch_in))
{
    // Assign the new switch value (input_value) to the
    // network variable (nv_switch_state); the NEURON CHIP
    // automatically sends out the new value over the
    // network.

    nv_switch_state = input_value ? ST_ON : ST_OFF;
}
```

Step 6: Install Nodes in a Development Network and Test

Once an individual node functions, connect it to another node and test it again. Gradually add new nodes to existing, functioning systems until you build a representative network, or test the necessary combinations of nodes.

In this example, if two switch nodes were connected to three light nodes for testing, the network would be configured as shown in Figure 1-5.

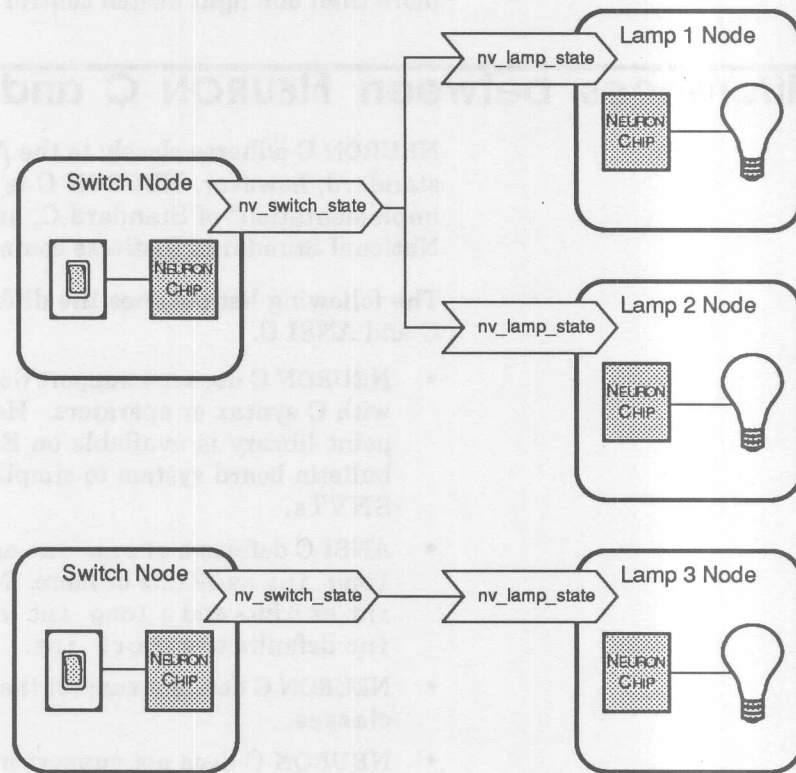


Figure 1-5. Sample Development Network with Five Nodes

The connections are specified by the person installing the network, and established by the network management tool. The network management tool generates network addresses and loads them over the network.

As the example shows, it is possible to connect more than one input network variable to the same output network variable. For example, it would be possible to have one light switch control multiple lights. It would also be possible to connect more than one output network variable to the same input network variable. For example, it would be possible to have more than one light switch control the same light.

Differences between NEURON C and ANSI C

NEURON C adheres closely to the ANSI C language standard; however, NEURON C is not a “conforming implementation” of Standard C, as defined by the American National Standards Institute committee X3-J11.

The following list outlines the differences between NEURON C and ANSI C.

- NEURON C does not support floating point computation with C syntax or operators. However, a simple floating point library is available on Echelon's LONLINK™ bulletin board system to simplify use of floating point SNVTs.
- ANSI C defines a short int as 16 bits or more and a long int as 32 bits or more. NEURON C defines a short int as 8 bits and a long int as 16 bits. In NEURON C, int defaults to a short int.
- NEURON C does not support the register or volatile classes.
- NEURON C does not support initializers in declarations of automatic variables.
- NEURON C does not support the address operator for automatic variables or function formal parameters.
- NEURON C does not support structures or unions as procedure parameters or as function return values.

constants (i.e. the contents of the variable referenced by the pointer can be read, but not modified).

- Macro arguments are not rescanned until after the macro is expanded, thus the macro operators '#' and '##' may not yield results as defined in the ANSI C standard when they occur in nested macro expansions.

NOTE: See the LONLINK bulletin board system for additional library functions including string and storage allocation.

- Names of network variables and message tags are limited to 16 characters.
- The only ANSI C library functions included in NEURON C are `memcpy()` and `memset()`. Other ANSI C library functions, such as `string`, `math`, `file` I/O, and storage allocation functions, are not included in NEURON C.
- The NEURON C implementation includes three ANSI include files: `STDDEF.H`, `STDLIB.H` and `LIMITS.H`.
- NEURON C requires use of the function prototype feature whenever a call to the function precedes the function definition (see Chapter 2).
- NEURON C contains additional reserved words and syntax not found in ANSI C. See Appendix B for the syntax summary and Appendix C for reserved words.
- NEURON C supports binary constants in addition to octal and hexadecimal. Binary constants are specified as `0b<binary_number>`. For example, `0b1101` equals decimal 13.
- NEURON C supports the `//` comment style from C++ in addition to the traditional `/* */` style. In the `//` style, two slashes (`//`) begin a comment. The comment is terminated by the end of the line, without further punctuation.

```
...C code  /* An ANSI C and NEURON C comment */
```

```
...C code  // A line-style NEURON C comment
```

- The `main()` construct is not used. Instead, a NEURON C program's executable objects consist of when statements in addition to functions. A thread of execution always begins with a when statement, as described in Chapter 2.
- NEURON C does not support multiple source files in separate compilation units (however, the `#include` directive is supported).

- The ANSI C preprocessor directives: #if, #elif, and #line are not supported.

NEURON C Variable Types

NEURON C supports the following C variable types:

[signed] long int	16-bit quantity
unsigned long int	16-bit quantity
signed char	8-bit quantity
[unsigned] char	8-bit quantity
[signed] [short] int	8-bit quantity
unsigned [short] int	8-bit quantity
enum(int type)	8-bit quantity

NEURON C provides one predefined enum :

```
typedef enum {FALSE, TRUE} boolean;
```

NEURON C also provides predefined types for I/O objects and for SNVTs. See Chapter 2 and Chapter 3 for more details.

NEURON C Storage Classes

If no class is specified and the declaration is at file scope, the data or function is global. (*File scope* is that part of a NEURON C program that is not contained within a function or a task.) Global data is present throughout the entire execution of the program. Upon power-up or reset of the NEURON CHIP, the global data is initialized to its initial-value expression, if present, otherwise to zero (variables declared with the eeprom or config class are only initialized when the application image is first loaded).

NEURON C supports the following ANSI C storage classes and type qualifiers:

auto	is a variable of local scope. Typically, this would be within a function body.
const	is a value which cannot be modified by the application program.
extern	is a data item or function that is defined in another module, in a library, or in the system firmware image.
static	is a data item or function which is not to be made available to other modules at link time. Furthermore, if the data item is local to a function or to a when task, the data value is to be preserved between invocations.

In addition to the ANSI C storage classes, NEURON C also provides the following classes:

NOTE: Use of `config` implies `const data`.

config	this class keyword can be combined only with an input network variable declaration, is located in EEPROM, and can be changed only by another node. A config network variable is used for application configuration.
network	this class keyword introduces a network variable. See Chapter 3 for more details.

system

this class keyword is used in NEURON C solely to access the NEURON CHIP firmware function library. The application programmer should not use this keyword for their data or function declarations.

The following keywords in NEURON C allow the programmer to direct portions of application code to specific memory sections:

- eeprom
- far
- ram

These keywords are particularly useful on the NEURON 3150 CHIP, since a majority of its address space is mapped off chip. See the *Memory Regions* section in Chapter 6 for a more detailed description of memory usage and these keywords and their uses.

Variable Initialization

Initialization of variables occurs at different times for different classes. The `const` variables, except for network variables, *must* be initialized. Initialization of `const` variables occurs when the application image is first loaded into the NEURON CHIP. The `eeprom` and `config` variables are also initialized at load time.

Global variables are initialized at reset (that is, when the node is reset or powered up). By default, all global variables are initialized to zero at this time. Initialization to zero costs no extra code space.

Initialization of I/O objects and input network variables (except for eeprom, config, or const network variables) also occurs at reset. Zero is the default value for network variables.

Integer Constant Typing

Decimal integer constants have the following default types:

0 .. 127	signed short
128 .. 32767	signed long
32768 .. 65535	unsigned long

The default type can be modified with the `u`, `U`, `l`, and `L` suffixes. For example:

0L	signed long
128U	unsigned short
128UL	unsigned long
256U	unsigned long

Hexadecimal, octal, and binary constants have the following default types, which can also be modified as described above with the `u`, `U`, `l`, and `L` suffixes:

0x0 .. 0x7f	signed short
0x80 .. 0xff	unsigned short
0x100 .. 0x7fff	signed long
0x8000 .. 0xffff	unsigned long

NEURON C Declarations

Both ANSI C and NEURON C support declarations of:

<i>Declaration</i>	<i>For Example:</i>
• Simple data items	<code>int a, b, c;</code>
• Data types	<code>typedef unsigned long ULONG;</code>
• Enumerations	<code>enum hue {RED, GREEN, BLUE};</code>
• Pointers	<code>char *p;</code>
• Functions	<code>int f(int a, int b);</code>
• Arrays	<code>int a[4];</code>
• Structures and unions	<pre>struct s { int field1; unsigned field2 : 3; unsigned field3 : 4; };</pre>

In addition, NEURON C supports declarations of:

<i>Declaration</i>	<i>For Example:</i>
• I/O objects	<code>IO_0 output oneshot relay_trigger;</code> (see Chapter 2)
• Timers	<code>mtimer led_on_timer;</code> (see Chapter 2)
• Network variables	<code>network input int temperature;</code> (see Chapter 3)

Compiler Directives

ANSI C permits compiler extensions through the `pragma` directive. These directives are implementation-specific.

Pragmas can be used to set certain NEURON CHIP system resources and node parameters such as buffer counts and sizes and receive transaction counts. See Chapter 6 for a detailed description of the compiler directives for buffer allocation.

Additional `pragma` directives can be used to control other NEURON CHIP-specific parameters. These directives can appear anywhere in the source file. The following pragmas are defined:

`#pragma all_bufls_offchip`

This pragma is only used with the LONBUILDER MIP/DPS. It causes the compiler to instruct the firmware and the linker to place all communication buffers in offchip RAM. This pragma is useful only on the NEURON 3150 CHIP. See the LONBUILDER Microprocessor Interface Program (MIP) User's Guide for more information.

`#pragma app_buf_in_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma app_buf_in_size`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma app_buf_out_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma app_buf_out_priority_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma app_buf_out_size`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma disable_mult_module_init`

specifies to the compiler to generate any required initialization code directly in the special init and event block, rather than as a separate procedure callable from the special init and event block. The in-line method, which is selected as a result of this pragma, is slightly more efficient in memory usage. This pragma should only be used when trying to shoehorn a program into a NEURON 3120 CHIP. See the discussion on *What to Try When a Program Doesn't Fit on a 3120* in Chapter 6.

`#pragma disable_servpin_pullup`

disables the internal pullup on the service pin. The pragma takes effect during I/O initialization. (This pullup is normally enabled.)

`#pragma disable_snvt_si`

disables generation of the Self-Identification (SI) data. The SI data is generated by default, but may be disabled using this pragma in order to reclaim program memory when the feature is not needed. This pragma may only appear once in the source program. See the *Standard Network Variable Types (SNVTs)* section in Chapter 3.

`#pragma enable_io_pullups`

enables the internal pullups on pins IO_4 through IO_7. The pragma takes effect during I/O initialization. (These pullups are normally disabled.) This pragma is useful because it can eliminate external components when pullups are required.

#pragma enable_multiple_baud

must be used when using multiple I/O devices which have differing bit rates. If needed, this pragma must appear prior to the use of any I/O function (e.g. `io_in()`, `io_out()`).

#pragma enable_sd_nv_names

causes the compiler to include the network variable names in the Self-Documentation (SD) information when Self-Identification (SI) data is generated. This pragma may only appear once in the source program. See the *Standard Network Variable Types (SNVTs)* section in Chapter 3.

#pragma explicit_addressing_off

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

#pragma explicit_addressing_on

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

#pragma fyi_off

causes the compiler to suppress informatory messages. Informatory messages are less severe than warnings, yet may indicate a problem in a program, or a place where code could be improved. Informatory messages are off by default at the start of compilation.

#pragma fyi_on

causes the compiler to issue informatory messages. This directive re-enables the printing of FYI messages after the `#pragma fyi_off` directive has been used. Informatory messages are off by default at the start of a compilation.

#pragma micro_interface

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

#pragma net_buf_in_count

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma net_buf_in_size`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma net_buf_out_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma net_buf_out_priority_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma net_buf_out_size`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma netvar_processing_off`

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

`#pragma netvar_processing_on`

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

`#pragma num_addr_table_entries nn`

sets the number of address table entries to *nn*. Valid values for *nn* are 0 to 15. The default number of address table entries is 15. This pragma is useful because it allows you to trade EEPROM space for address table size (see Chapter 6).

`#pragma num_domain_entries nn`

sets the number of domain table entries to *nn*. Valid values for *nn* are 1 to 2. The default number of domain table entries is 2. This pragma is useful because it allows you to trade EEPROM space for domain table size (see Chapter 6).

`#pragma one_domain`

allows a NEURON CHIP to be configured for one domain only. The default is two domains. This pragma is useful because it frees up EEPROM space when only one domain is required (see Chapter 6).

`#pragma ram_test_off`

disables the testing of off-chip RAM buffer space to speed up initialization. Normally the first thing the NEURON CHIP firmware does when it comes up after a reset or power-up is to verify basic functions such as CPUs, RAM and timer/counters. This can consume large amounts of time, particularly at slower clock speeds. By turning off RAM buffer testing, you can trade off some reset time for maintainability. All static variables are nevertheless initialized to zero.

`#pragma read_write_protect`

allows a node's program to be read and write protected to prevent copying or alteration via the network. This feature provides protection of a manufacturer's confidential algorithms. A node cannot be reloaded once it is protected. The write protection feature is included to disallow "trojan horse" intrusions. The protection must be specifically enabled in the NEURON C source program. Once a custom node has been loaded with an application containing this pragma, the application program can never be reloaded on a NEURON 3120 CHIP. It is possible, however, on the NEURON 3150 CHIP, with the use of the EEBLANK program. See the *Building Application Nodes* section in Chapter 7 of the *LONBUILDER User's Guide*. This applies equally to custom nodes in a development network. An appropriate error message is printed by the LONBUILDER network manager when a load fails due to write protection. For LONBUILDER SBCs on the network, the SBC can be placed on the LONBUILDER backplane for reinstalling and reloading.

`#pragma receive_trans_count`

See the *Allocating Buffers* section in Chapter 6 for more detailed information on this pragma and its use.

`#pragma scheduler_reset`

causes the scheduler to be reset within the nonpriority when clause execution cycle, after each event is processed (see Chapter 5).

`#pragma set_id_string "ssssssss"`

provides a mechanism for using the node's 8-byte identification string. It initializes the 8-byte program ID string located in the application image. The ID string is sent as part of the service pin message transmitted when the service pin is activated. The ID string may be set to any C string constant, 8 characters or less. The uppermost bit in the first character cannot be set. If this pragma is used for a given node, the directive `#pragma set_std_prog_id` cannot be used. Neither pragma is required.

`#pragma set_netvar_count nn`

This pragma is only used with the LONBUILDER Microprocessor Interface Program. See the *LONWORKS Network Interface Developer's Guide* for more information.

`#pragma set_node_sd_string <C string constant in quotes>`

specifies and controls the generation of Self-Documentation (SD) data in a node's application image. The node as a whole may have an SD string. This string defaults to a NULL string and may have a maximum of 255 bytes. This pragma may be used to explicitly set this string. Concatenated string constants are not allowed. This pragma may only appear once in the source program.

`#pragma set_std_prog_id hh:hh:hh:hh:hh:hh:hh:hh`

provides a mechanism for using the node's 8-byte identification string. It initializes the 8-byte node identification string using the hexadecimal values given (*hh* is a hexadecimal number from 0 to ff). The first byte can only have values from 80 .. 8f. If this pragma is used for a given node, the directive `#pragma set_id_string` cannot be used. Neither pragma is required.

Table 1-1 and Figure 1-7 show the standard program identification fields that form this 8-byte string.

Table 1-1. Standard Program Identification Fields

<i>Field</i>	<i>Size</i>	<i>Type</i>	<i>Assigned By</i>
format	4 bits	unsigned	Echelon
manufacturer ID	20 bits	unsigned	Echelon
device class	16 bits	unsigned	Echelon
device subclass	16 bits	unsigned	Echelon
model number	8 bits	unsigned	manufacturer

Program ID formats 8 and 10-15 are reserved for interoperable LONMARK nodes. Format 9 can be used during development to test decoding of standard IDs by a network management tool.

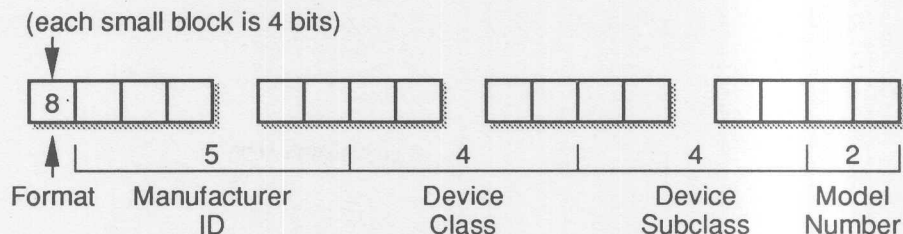


Figure 1-6. Diagram of Standard Program Identification Fields

■ Page 1-35 Additions

Add the following pragmas after Figure 1-6.

`#pragma snvt_si_eecode` causes the compiler to force the linker to locate the SNVT Self/Identification information in EECODE space. (By default, the linker *may* place the table in EEPROM or in ROM code space, as it determines.) Placing this table in EEPROM ensures that it may be modified via network management memory write commands. A network management tool can use this capability to modify the standard network variable types (SNVTs) on a node during installation. This is useful for nodes that may be connected to different types of I/O devices, and is also useful for data loggers that can collect data from a variety of nodes. This pragma is only useful on a NEURON 3150 CHIP.

`#pragma snvt_si_ramcode` causes the compiler to force the linker to locate the SNVT Self/Identification information in RAMCODE space. (By default, the linker *may* place the table in EEPROM or in ROM code space, as it determines.) Placing this table in RAM ensures that it may be modified via network management memory write commands. **NOTE: RAMCODE space is always external memory, and is assumed to be non-volatile.** This pragma is only useful on a NEURON 3150 CHIP. See `#pragma snvt_si_eecode` for an example of usage.

`#pragma warnings_off` causes the compiler to suppress warning messages. Warning messages generally indicate a problem in a program, or a place where code could be improved. Warning messages are on by default at the start of a compilation.

`#pragma warnings_on` causes the compiler to issue warning messages. This directive re-enables the printing of warning messages after the `#pragma warnings_off` directive has been used. Warning messages are on by default at the start of a compilation.

2

Focusing on a Single Node

This chapter describes the NEURON CHIP event scheduler and I/O objects. The concept of *predefined events* and *user-defined events* is introduced. Objects that can be defined for each NEURON CHIP include *timers* and *input/output (I/O) objects*, described here, *network variables*, described in Chapter 3, *How Nodes Communicate Using Network Variables*, and *explicit messages*, described in Chapter 4, *How Nodes Communicate Using Explicit Messages*. Code examples in this chapter illustrate the use of events, I/O and timer objects, and I/O functions.

What Happens on a Single Node?

In this chapter, you begin to learn about programming the NEURON CHIP by focusing on a single node. Each NEURON CHIP has its own scheduler, timers, and logical I/O devices. NEURON C includes predefined objects that provide access to these devices. These objects are described briefly here, and in detail later in this chapter.

- The NEURON CHIP *event scheduler* handles task scheduling for the NEURON CHIP. This chapter explains how to define events and tasks, how the scheduler evaluates nonpriority events, and how you can define priority events.
- The NEURON CHIP offers two types of *timer* objects: millisecond and second timers. These timers can be used to affect the scheduling of tasks, as described in the section on *Timers*.
- A number of *I/O objects* can be declared using NEURON C extensions to ANSI C. These I/O objects, as well as related I/O functions and events, are described in detail in the section on *Input/Output*.

The Scheduler

The scheduling of NEURON CHIP tasks is event driven: when a given condition becomes TRUE, a body of code (called a *task*) associated with that condition is executed. The scheduler allows you to define tasks that run as the result of certain events, such as a change in the state of an input pin, receiving a new value for a network variable, or the expiration of a timer. You can also specify certain tasks as *priority* tasks, so that they receive preferential service. (See the *Priority When Clauses* section later in this chapter.)

When Clauses

Events are defined through when clauses. A when clause contains an expression which, if evaluated as TRUE, causes the body of code (the *task*) following the expression to be

executed to completion. Here's a simple when clause and its associated task:

```
when (timer_expires (led_timer)) ← when clause
{
    //Turn off the LED
    io_out (io_led, OFF): ← task
}
```

In this example, when the application timer `led_timer` (defined elsewhere) expires, the body of code (the task) that follows the when clause is executed to turn off the specified I/O object, `io_led` (also defined elsewhere in the program). After this task has been executed, the `timer_expires` event is cleared. Its task is then ignored until the LED timer expires again and the when clause again evaluates to TRUE.

Multiple when clauses can be associated with a single task. The following is for example purposes only:

```
when (reset)
when (io_changes(io_switch))
when (!timer_expires)
when (flush_completes && (y == 5))
when (x == 3)
{
    // Turn on the LED and start the timer
    .
    .
    .
}
```

When clauses cannot be nested. For example, the following nested when clause is not valid:

```
when (io_changes(io_switch))
{
    when (x == 3) { // Can't nest!
        ...
    }
}
```

Also refer to Figure 5.1, *NEURON CHIP Firmware Scheduling of Regular and Priority when Clauses*, in Chapter 5.

When Statement

The syntax for a when statement (the when clause or clauses plus associated task) is:

```
when-clause  
[when-clause ... ]  
task
```

when-clause

The syntax for *when-clause* is:

```
[priority] when (event)
```

priority

is an option used to force evaluation of the following when clause each time the scheduler runs. See the *Priority When Clauses* section later in this chapter.

event

is either a predefined event (see the following section) or any valid NEURON C expression (which may contain a predefined event). Predefined events as well as expressions are enclosed in parentheses.

One or more when clauses can be associated with the same task.

task

is a NEURON C compound statement, consisting of a series of NEURON C declarations and statements, enclosed in braces, which are identical to those found in a NEURON C function definition. The task is identical to the body of a void function (that is, it cannot return a value). A `return;` statement can be used to terminate execution of the task but is not required.

Types of Events Used in When Clauses

The events defined in a when clause fall into two general categories: predefined events and user-defined events. *Predefined events* use keywords built into the compiler. Examples of predefined events include input pin state changes, network variable changes, timer expiration, and message reception. *User-defined events* can be any valid NEURON C expression.

The distinction between user-defined events and predefined events is not critical. Use predefined events whenever possible, since they require less code space.

There is one exception to the statement that a when clause can be any valid C expression. The predefined events 'offline', 'online' and 'wink' must appear by themselves if used. All other predefined events may be combined into any arbitrary expressions. This restriction only applies to when clauses.

Examples

```
when (msg_arrives)                // O.K.
when (msg_arrives && flag == TRUE) // O.K.
when (online)                      // O.K.
when (online && flag == TRUE) // Not permitted.
```

Predefined Events

The `timer_expires` event shown earlier is one type of predefined event. Other predefined events are represented by unique keywords, listed here. Some predefined events, such as the I/O events and network variable events, may be followed by a modifier that narrows the scope of the event. If the modifier is optional and not supplied, any event of that type qualifies. The predefined events are:

<i>Predefined Event</i>	<i>Where Described in This Manual</i>
<code>flush_completes</code>	Chapter 5
<code>io_changes</code>	this chapter
<code>io_in_ready</code>	this chapter
<code>io_out_ready</code>	this chapter
<code>io_update_occurs</code>	this chapter
<code>msg_arrives</code>	Chapter 4
<code>msg_completes</code>	Chapter 4
<code>msg_fails</code>	Chapter 4
<code>msg_succeeds</code>	Chapter 4
<code>nv_update_occurs</code>	Chapter 3
<code>nv_update_completes</code>	Chapter 3
<code>nv_update_fails</code>	Chapter 3
<code>nv_update_succeeds</code>	Chapter 3
<code>offline</code>	Chapter 5
<code>online</code>	Chapter 5
<code>reset</code>	following paragraphs
<code>resp_arrives</code>	Chapter 4
<code>timer_expires</code>	this chapter
<code>wink</code>	Chapter 5

Predefined events can also be used as any sub-expression, including within the control expression of `if`, `while`, and `for` statements. This method is termed *direct event processing*. An example of direct event processing is:

```

mtimer t;
when (event)
{
    .
    .
    if (timer_expires(t)) {
        io_out(io_led, OFF);
    }
    .
    .
}

```

Any built-in event keyword or keyword expression (e.g. `timer_expires(t)`) will be treated the same as any other sub-expression and any combination allowed by standard C expression syntax is allowed when programming in NEURON C.

The special case of the `io_changes` event expression must be treated carefully. The `to` and `by` qualifier keywords are treated as general expression operators for purposes of precedence (although they are only permitted in combination with `io_changes`). These operators are of equal precedence with each other (they are mutually exclusive), they are of higher precedence than relational operators (i.e. comparisons), but lower in precedence than shift and arithmetic operators.

Following are examples of how the `io_changes` event expression is parsed:

`io_changes (device) by a + b`

as:

`io_changes (device) by (a + b)`

and

`io_changes (device) by a < b`

as:

`(io_changes (device) by a) < b`

As with any other C operators, the implied precedence can be explicitly changed by parenthesization.

The NEURON C compiler detects the use of predefined event keywords in when clauses and treats them specially for code optimization purposes. However, when event keywords are used as subexpressions within when clauses, event table optimizations cannot be used. In the examples below, the first case uses the event table optimization, the second and third do not:

```
when (timer_expires) {}  
when (! timer_expires) {}  
if (timer_expires)
```

Although the `io_changes` expression (by and to varieties) does not require a constant value, only constant-valued `io_changes` expressions are optimized into the when clause event table.

Reset Event

The reset event is TRUE the first time this event is evaluated after the NEURON CHIP is reset for any reason. (I/O object and global variable initializations will be performed before processing any events.) The reset event task is the first task to be executed first after reset of the NEURON CHIP.

User-defined Events

A user-defined event can contain assignments and function calls. Calls to complex functions should be used cautiously, because they impact response time for all events within the program. Assignments within user-defined events can only be done to global variables.

Event Processing

Events related to network activity are processed using two separate queues. One queue serves events related to incoming network messages (`nv_update_occurs`, `msg_arrives`, `online`, `offline`, and `wink`) while the other queue serves the remaining network events pertaining to completion events and responses. Most network events, except `resp_arrives`, `online`, `offline`, and `wink`, are enqueued only if the NEURON C compiler has determined that the application checks for the event. The `online`, `offline`, and `wink` events are always enqueued but are discarded by the scheduler if no corresponding when clause is found.

Once at the head of the queue, an event remains there until processed by the application. Therefore, any network event which is checked for by an application must be checked for frequently or the event may remain at the head of the queue, effectively blocking that queue. A blocked queue prevents the application from continuing normal processing of events and can cause the node to fail to respond to any subsequent application or network management messages. This is particularly critical for `nv_update_occurs` and `msg_arrives` events which can arrive unsolicited, at any time; in comparison, completion events and responses arrive only as the result of application-initiated outgoing network activity. The NEURON C compiler will determine that an event is handled by the application by virtue of its presence in the program, even if it is never checked for in a when clause, or is only checked for in special circumstances.

will clear any pending event, regardless of whether the entire expression is TRUE or FALSE, as below:

```
when ((timer_expires(t)) && (flag == TRUE))
```

Scheduling of When Clauses

The scheduler evaluates when clauses in round-robin fashion: each when clause is evaluated by the scheduler and, if TRUE, the task associated with it is executed. If the when clause is FALSE, the scheduler moves on to examine the following when clause. After the last when clause, the scheduler returns to the top and moves through the group of when clauses again. For example, a group of when clauses might look like this:

```
when (nv_update_occurs)
    // {task to execute}
```

```
when (nv_update_fails)
    // {task to execute}
```

```
when (io_changes)
    // {task to execute}
```

```
when (timer_expires)
    // {task to execute}
```

For purposes of the following explanation, letter names are used for the clauses:

```
when (A)
```

```
when (B)
```

```
when (C)
```

```
when (D)
```

The following narration of events shows how the order of execution of tasks differs from the order of the when clauses themselves.

- 1 Only C is TRUE.
- 2 The scheduler begins with A. Since A is FALSE, its task is ignored.
- 3 The scheduler moves to B. Since B is FALSE, its task is ignored.
- 4 A becomes TRUE.
- 5 The scheduler moves to C. Since C is TRUE, its task is executed.
- 6 The scheduler moves to D. Since D is FALSE, its task is ignored.
- 7 The scheduler moves back to A. Since A is TRUE (item 4, above), its task is executed.

Priority When Clauses

The *priority* option can be used to designate when clauses that should be evaluated more often than nonpriority when clauses. Priority when clauses are evaluated in the order specified *every* time the scheduler runs. If any priority when clause evaluates to TRUE, the corresponding task is executed and the scheduler starts over at the top of the priority when clauses.

NOTE: Use of the priority when clause should be carefully considered, since too many priority when clauses might starve execution of nonpriority when clauses. In addition, if a priority when clause is true the majority of the time, it monopolizes processor time.

If none of the priority when clauses evaluate to TRUE, then a nonpriority when clause is evaluated, selected in the round-robin fashion described earlier. If the selected nonpriority when clause evaluates to TRUE, its task is executed. The scheduler then resumes with the first priority when clause. (If the nonpriority when clause selected evaluates to FALSE, its task is ignored and the scheduler resumes with the first priority when clause. See Figure 5.1 in Chapter 5.)

The scheduling algorithm described above can be modified through use of the `scheduler_reset` pragma, discussed in the *Additional Features* section in Chapter 5.

Function Prototypes

NEURON C requires use of function prototypes if a function is to be called before it is defined. Examples of valid prototypes include the following:

```
void f(void);
int g(int a, int b);
```

The following are not considered prototypes. They are merely forward declarations:

```
void f();
g(); // defaults to 'int' return
```

If you define a function before you call it, NEURON C automatically creates an internal prototype for you. Only one prototype is created for a given function. The following examples are technically not prototypes, but NEURON C creates function prototypes for them:

```
void f()
{ /* body */ }

g (a,b)
int a;
int b;
{ /* body */ }
```

Although NEURON C can create prototypes, it does *not* employ the ANSI C Miranda prototype rule. (According to the Miranda prototype rule, if a function call does not already have a prototype, a prototype will automatically be created for it.) In NEURON C, a function prototype is automatically created only if the function has been previously defined.

Timers

Two types of timer objects are available to a NEURON C application: millisecond and second timers. The millisecond timer provides a timer duration of 1 to ~~65,535~~ ⁶⁴⁰⁰⁰ milliseconds (or .001 to ~~65.535~~ seconds). The second timer provides a timer duration of 1 to 65,535 seconds. For more accurate second timing, use the millisecond timer (See also the *Input Clock Frequency* and *Timer Accuracy* sections later in this chapter.)

Declaring Timers

A maximum of 15 timer objects (total of both types) can be defined within a single program. A timer object is declared using one of the following:

mtimer [repeating] timer-name;

stimer [repeating] timer-name;

mtimer indicates a millisecond timer.

stimer indicates a second timer.

repeating is an option for the timer to restart itself automatically upon expiration. With this option, accurate timing intervals can be maintained even if the application cannot respond immediately to an expiration event.

timer-name is a user-supplied name for the timer. Assigning a value to this name starts the timer for the specified length of time (the specified time is in seconds for an stimer and milliseconds for an mtimer). A timer that is running or has expired can be started over by assigning a new value to this name. The timer object can be

evaluated while the timer is running, and it will indicate the time remaining. Setting the timer to 0 turns the timer off. No timer expiration event occurs for a timer that has been turned off (see the description of the `timer_expires` Event described in *Section C.1* in Appendix C).

An example of declaring a timer object and assigning a value to it is:

```
stimer led_timer;
when (reset)
{
    led_timer = 5; // start timer with
                  // value of 5 sec
}
```

An example of turning a timer off is

```
stimer led_timer;
when (t == 50)
{
    led_timer = 0;
}
```

NOTE: When setting and examining timers used in the NEURON C debugger, certain inaccuracies may occur. When a timer is set during program execution and is examined while the program is halted (includes single stepping and breakpoints), the timer value can be as much as 200 milliseconds larger than the actual time until expiration. No such inaccuracy exists on a timer which is allowed to run without a debugger halt.

An example of evaluating the value of a running timer is

```
stimer led_timer;
when (nv_update_occurs)
{
    time_remaining = led_timer;
    .
    .
    .
}
```

timer_expires Event

The `timer_expires` event becomes TRUE when a timer expires. The syntax of this event is:

timer_expires [(*timer-name*)]

timer-name is an option to specify a specific timer to check

NOTE: Be sure to check for specific timer events while using the unqualified `timer_expires` event. An unqualified `timer_expires` event remains TRUE as long as any timer has expired.

If the *timer_name* option is not included, the event is an unqualified `timer_expires` event. Unlike all other predefined events, which are TRUE only once per pending event, the unqualified `timer_expires` event remains TRUE as long as *any* timer has expired. This event can be cleared only by checking for specific timer expiration events. For example, the following when clause checks for the expiration of the LED timer, so the `timer_expires` event is cleared to FALSE.

```
stimer led_timer;
when (timer_expires(led_timer))
{
    // Turn off the LED
    io_out(io_led, OFF);
}
```


If your program has multiple timers, include a specific check for each timer so that the expiration event is cleared, as follows:

```
mtimer x;  
mtimer y;  
mtimer z;  
  
when (timer_expires(x))  
{  
    // task  
}  
  
when (timer_expires(y))  
{  
    // task  
}  
  
when (timer_expires(z))  
{  
    // task  
}
```

An alternate way to check for specific timers would be as follows:

```
when (timer_expires)  
{  
    if (timer_expires(x))  
        .  
        .  
        .  
    else if (timer_expires(y))  
        .  
        .  
        .  
    else if (timer_expires(z))  
        .  
        .  
        .  
}
```

Which style you choose depends on the circumstances. Use the first style of checking for specific timers if you're concerned about code space. Use the second style if you're concerned about speed of execution, performance, or response time.

For an example of a complete program that declares a timer and uses the `timer_expires` event, see the *Thermostat Example* later in this chapter.

Input/Output

The NEURON CHIP has a variety of built-in electrical interface options for performing input and output (I/O) functions. Before performing I/O, you must first declare the I/O objects that interface with the eleven NEURON CHIP I/O pins, named `IO_0`, `IO_1`, ..., `IO_10`. Any undeclared pin is, by default, unused and thus deactivated. In the deactivated state, the pin is in a high-impedance (tri-state) condition. The declaration syntax for I/O objects is described in detail in Appendix C.

To perform I/O, use the built-in I/O functions `io_in()`, `io_out()`, `io_select()`, `io_change_init()`, and `io_set_clock()`. The `io_out_request()` function is used to perform I/O with the parallel I/O object. Use of these I/O functions is described in this chapter.

I/O objects can also be linked to NEURON C events, since changes in I/O often affect task scheduling. See the section later in this chapter on *I/O Events* for a description of `io_changes` and `io_update_occurs`, which are the I/O-related events used in `when` clauses. For more detailed information on, and additional examples of, using I/O, see the following LONWORKS Engineering Bulletins:

- *Analog-to-Digital Conversion with the NEURON CHIP* Engineering Bulletin (part no. 005-0019-01)
- *Driving a Seven Segment Display with the NEURON CHIP* Engineering Bulletin part no. 005-0014-01)
- *NEURON CHIP Quadrature Input Function Interface* Engineering Bulletin (part no. 005-0003-01)
- *Parallel I/O Interface to the NEURON CHIP* Engineering Bulletin (part no. 005-0021-01)
- *RS-232C Serial Interfacing with the NEURON CHIP* Engineering Bulletin (part no. 005-0008-01)

I/O Object Types

A variety of I/O object types can be defined using NEURON C objects. Object types can be grouped as follows:

- *Direct I/O Objects* are based on a logic level at the I/O pins; none of the NEURON CHIP's hardware timer/counters are used in conjunction with these I/O objects. These objects can be used in multiple, overlapping combinations within the same NEURON CHIP. Direct I/O object types are:

<i>Input Object Type</i>	<i>Output Object Type</i>
bit	bit
bitshift	bitshift
byte	byte
nibble	nibble
leveldetect	

- *Timer/Counter I/O Objects* use a timer/counter circuit in the NEURON CHIP. A NEURON CHIP has two timer/counter circuits, one whose input can be multiplexed and one with dedicated input. Timer/counter I/O object types are:

<i>Input Object Type</i>	<i>Output Object Type</i>
pulsecount	pulsecount
ontime	frequency
period	oneshot
quadrature	pulsewidth
totalcount	triac
	triggeredcount

- *Serial I/O Objects* are used for transferring data serially over a pin or set of pins. Only one type of serial I/O object can be defined within a NEURON CHIP. Both the input and output versions of the serial type can coexist within a single NEURON CHIP. Serial I/O object types are:

<i>Input Object Type</i>	<i>Output Object Type</i>
serial	serial
<i>Input / Output Object Type</i>	
neurowire	

See the *RS-232C Serial Interfacing with the NEURON CHIP* Engineering Bulletin (part number 005-0008-01) and *Driving a Seven Segment Display with the NEURON CHIP* Engineering Bulletin (part no. 005-0014-01) for more information.

- The *Parallel I/O Object* is used for high-speed bidirectional I/O. This object type uses all the NEURON CHIP I/O pins. The parallel I/O object type is:

<i>Input / Output Object Type</i>
parallel

See the *Parallel I/O Interface to the NEURON CHIP* Engineering Bulletin (part no. 005-0021-01) for more information.

Table 2-1 lists the object types, which pins they can use, and what additional options apply to them. The `io_out()` call has an unsigned short return value signifying the number of bits actually transferred if the object is of type neurowire slave. Else, the return value is void. Also see Appendix C.

Table 2-1. IO Object Types (Part 1 of 2)

Object Type	Available Objects	as First Pin/ Total # Pins per Object	Other Options
Bit input	11	any pin / 1 pin	
Bit output	11	any pin / 1 pin	initial_output_level

■ Pages 2-19 and 2-20 Replace table 2-1 with the new table below.

Table 2-1. IO Object Types (Part 1 of 2)

Object Type	Max # Available Objects	Pins Declarable as First Pin/ Total # Pins per Object	Other Options
Bit input	11	any pin / 1 pin	
Bit output			initial_output_level
Bitshift input	5	IO_0 - IO_6, IO_8, or IO_9 / 2 pins	numbits, clockedge, kbaud
Bitshift output	5	IO_0 - IO_6, IO_8, or IO_9 / 2 pins	numbits, clockedge, kbaud, initial_output_level
Byte input	1	IO_0 / 8 pins	-
Byte output	1	IO_0 / 8 pins	initial_output_level
Dualslope input	2	IO_4-IO_7 IO_0 is used when input is IO_4; (mux) or IO_5-IO_7; IO_1 is used when input is IO_4 (ded)/2 pins	invert, clock
Edgelog input	1	IO_4/1 pin	invert, clock
Frequency output	2	IO_0 or IO_1 / 1 pin	invert, clock, initial_output_level
Infrared input	4	IO_4-IO_7/1 pin	invert, clock
Leveldetect input	8	IO_0 - IO_7 / 1 pin	-
Magcard input	1	IO_8, uses 2 pins; timeout pin is IO_0-IO_7, 1 pin/a total of 3 pins	timeout pin
Muxbus input/output	1	IO_0/11 pins	
Neurowire master input/output	8	IO_8, uses 3 pins; select pin is IO_0 - IO_7, 1 pin / a total of 4 pins	select pin, kbaud
Neurowire slave input/output	1	IO_8, uses 3 pins; timeout pin is IO_0 - IO_7, 1 pin / a total of 4 pins	timeout pin
Nibble input	2	IO_0 - IO_4 / 4 pins	
Nibble input	2	IO_0 - IO_4 / 4 pins	initial_output_level
Oneshot output	2	IO_0 or IO_1 / 1 pin	invert, clock, initial_output_level
Ontime input	4	IO_4 - IO_7 / 1 pin	mux ded, invert, clock

Table 2-1. I/O Object Types (Part 2 of 2)

Object Type	Max # Available Objects	Pins Declarable as First Pin/ Total # Pins per Object	Other Options
Parallel input/output	1	IO_0 / 11 pins	slave slave_b master
Period input	4	IO_4 - IO_7 / 1 pin	mux ded, invert, clock
Pulsecount input	4	IO_4 - IO_7 / 1 pin	mux ded, invert
Pulsecount output	2	IO_0 or IO_1 / 1 pin	invert, clock
Pulsewidth output	2	IO_0 or IO_1 / 1 pin	invert, clock, short, long initial_output_level
Quadrature input	2	IO_4 or IO_6 / 2 pins	-
Serial input	1	IO_8 / 1 pin	baud
Serial output	1	IO_10 / 1 pin	baud
Totalcount input	4	IO_4 - IO_7 / 1 pin	mux ded, invert
Triac output	2	IO_0 or IO_1; sync pin can be IO_4 - IO_7 when IO_0 is output pin; sync pin is IO_4 when IO_1 is output pin / 2 pins	sync pin, invert, clock, clockedge
Triggeredcount output	2	IO_0 or IO_1; sync pin can be IO_4 - IO_7 when IO_0 is output pin; sync pin is IO_4 when IO_1 is output pin / 2 pins	sync pin, invert

Declaring an I/O object accomplishes two things:

- 1 The declaration tells what type of I/O operation will be performed and on which pin or pins. Internal hardware within the NEURON CHIP is configured every time the NEURON CHIP is reset as a result of this declaration.
- 2 The declaration associates the name of the I/O object with the hardware.

This section describes the general syntax for declaring I/O objects. A detailed explanation of the syntax for each *type* can be found in Appendix C.

pin type [options] io_object_name;

pin

specifies one of the eleven I/O pins, IO_0 through IO_10. In general, pins can appear in a single object declaration only. However, a pin may appear in multiple declarations of the I/O object types bit, nibble, and byte. In this case, it is not required that all declarations have the same sense, that is, input versus output. See the *Overlaying I/O Objects* section later in this chapter.

type

specifies the I/O object type.

options

are optional I/O parameters. The particular option(s) available are described for each object type in Appendix C. Except where noted, these options can be listed in any order. All options have default values that are used when you do not include the option in the object declaration.

io_object_name

is a user-supplied name for the I/O object, in the ANSI C format for variable identifiers.

In this example, a logic level needs to be measured at an input pin, IO_3. The pin is connected to a proximity detector, as its name indicates:

```
IO_3 input bit io_prox_detector;
```

Now, whenever your program refers to `io_prox_detector`, it is actually referring to the logical level on pin IO_3.

Use of I/O Resources

The following list and Table 2-2 contain guidelines for declaring I/O object types:

- Up to 16 I/O objects can be declared.
- Timer/counter 1 can be multiplexed for up to four input objects.
- Neurowire, parallel, and serial I/O objects are exclusive. Only one member of this group of objects can be declared in one program.

• Because the parallel and muxbus I/O objects require all I/O pins, no other object types can be declared when either of these objects are declared.

- Direct I/O object types (bit, nibble, byte) can be declared in any combination (see the following section, *Overlaying I/O Objects*). Timer/counter, serial, and Neurowire I/O object declarations override the pin directions of any overlaying direct I/O object types.

• Quadrature and dualslope input objects cannot be multiplexed with other input objects on timer/counter 1. Edgelog input uses both timer/counters and is exclusive of any other timer/counter objects.

- Bitshift I/O objects cannot be declared on the same I/O pins as timer/counter objects. Direct I/O objects may be overlayed with bitshift I/O objects. Two adjacent bitshift I/O objects may not share any I/O pins.

As an example, the following I/O object types can be combined on the NEURON CHIP:

1 to 4 timer/counter inputs (multiplexed on IO_4, IO_5, IO_6, IO_7), including quadrature input on IO_6

or

1 timer/counter output (on IO_0)

and

1 timer/counter input (on IO_4), including quadrature input on IO_4

or

1 timer/counter output (on IO_1)

and

1 Neurowire I/O object type (on IO_8, IO_9, IO_10) and 1 of IO_0 through IO_7)

or

1 serial I/O object type (on IO_8, IO_10)

and

any direct I/O object type on any pin (IO_0 through IO_10)

or

1 parallel I/O object type (on IO_0)

or

1 muxbus I/O object type (on IO_0)

■ Page 2-24 Replace table with new table below.

Table 2-2. I/O Devices

		0	1	2	3	4	5	6	7	8	9	10
DIRECT I/O MODES	Bit Input, Bit Output											
	Byte Input, Byte Output											
	Leveldetect Input											
	Nibble Input, Nibble Output											
PARALLEL I/O MODES	Muxbus I/O											
	Parallel I/O { Master/Slave A											
	Slave B											
SERIAL I/O MODES	Magcard Input											
	Bitshift Input, Bitshift Output											
	Neurowire I/O { Master											
	Slave											
TIMER/COUNTER INPUT MODES	Serial Input											
	Dualslope Input											
	Edgeslog Input											
	Infrared Input											
TIMER/COUNTER OUTPUT MODES	Onetime Input											
	Period Input											
	Pulsecount Input											
	Quadrature Input											
	Totalcount Input											
	Frequency Output											
	Oneshot Output											
	Pulsecount Output											
	Pulsewidth Output											
	Triac Output											
	Triggeredcount Output											

Bitshift, Neurowire: C = Clock D = Data

Timer/Counter 1 Devices

- either: IO_6 input quadrature
- or: IO_4 input edgeslog
- or: IO_0 output [triac | triggeredcount] sync(IO_4...IO_7)
- or: IO_0 output [frequency | oneshot | pulsecount | pulsewidth]
- or: up to four of:
 - IO_4 input [ontime | period | pulsecount | totalcount | dualslope] mux
 - IO_5...IO_7 input [ontime | period | pulsecount | totalcount | dualslope]
- or: IO_4 input edgeslog

Timer/Counter 2 Devices

- either: IO_4 input quadrature
- or: IO_4 input edgeslog
- or: IO_1 output [triac | triggeredcount] sync(IO_4)
- or: IO_1 output [frequency | oneshot | pulsecount | pulsewidth]
- or: IO_4 input edgeslog
- or: IO_4 input [ontime | period | pulsecount | totalcount | dualslope] ded

Overlaying I/O Objects

In some cases, you may choose to declare more than one I/O object type for the same pin. For example, the following declarations allow a program to read four adjacent pins in one operation (with the nibble object type) or read each pin individually (with the bit object type):

```
IO_4 input nibble io_all_points;  
IO_4 input bit io_point_1;  
IO_5 input bit io_point_2;  
IO_6 input bit io_point_3;  
IO_7 input bit io_point_4;
```

The following declarations enable a program to monitor (read back) the level on its own oneshot output object:

```
IO_1 output oneshot clock (3) io_break_high;  
IO_1 input bit io_break_high_level;
```

Object types can be divided into two categories with respect to overlaying: hard and soft pin direction objects. A “hard” pin direction I/O object is not affected by subsequent declarations. “Soft” pin direction I/O objects (the bit, nibble, and byte object types) are changed by subsequent pin declarations. When multiple soft pin direction I/O objects are declared for the same pin, the last soft I/O object declared is the one that affects the initial direction of the pin at run-time.

The `io_set_direction()` function allows the application to change the direction of any bit, nibble, or byte type I/O pin at run time. See Appendix C for documentation of `io_set_direction()`.

Note in the previous example that oneshot is a hard pin direction I/O object, but bit is a soft pin direction I/O object. The order of declarations is not important, and the oneshot object is the one that affects the direction of pin IO_1 at run-time.

If a program declares the following

```
IO_2 input bit io_point_1;  
IO_2 output bit io_point_2;
```

pin IO_2 is an output bit I/O object (since it is last). A subsequent call to `io_out()` sets the level of this pin. An `io_in()` call can then be used to read back the actual pin level of this output object.

Performing I/O: Functions and Events

Input objects can be accessed in two ways: by using the explicit `io_in()` function, or by referring to an event associated with the object in a `when` clause. The following sections describe both methods.

I/O Functions

After you have declared the I/O objects for a NEURON CHIP, you can access the objects through the I/O functions provided in NEURON C. These functions are built into the NEURON C compiler and do not need to be declared or linked in. The compiler enforces type checking on the parameters of these functions.

■ Page 2-26 Additions

Replace the current I/O functions list with the following:

<code>io_change_init()</code>	initializes an object for the <code>io_changes</code> event
<code>io_edgelog_preload()</code>	sets timer/counter preload value
<code>io_in()</code>	reads data from an I/O object
<code>io_in_ready()</code>	event function which evaluates to TRUE when a block of data is available to be read from a parallel I/O object
<code>io_in_request()</code>	starts an I/O input cycle for the dualslope I/O object
<code>io_out()</code>	writes data to an I/O object
<code>io_out_request()</code>	request the write token for a parallel I/O object
<code>io_preserve_input()</code>	causes the first value obtained from a timer/counter after reset or an <code>io_select()</code> to be considered valid
<code>io_select()</code>	selects one of the multiplexed input objects (See the <i>I/O Multiplexing</i> section later in this chapter.)
<code>io_set_clock()</code>	changes the clock setting for the specified object
<code>io_set_direction()</code>	changes the direction of any bit, nibble, or byte type I/O pin(s)

Refer to the Appendix C changes in this supplement and to page C-24 and the function directory in the *NEURON C Programmer's Guide* for more information.

io_in() Function

The syntax for *io_in()* is:

```
return_value = io_in ( io_object_name [, args] )
```

io_object_name is the name for the I/O object, which corresponds to *io_object_name* in the I/O declaration.

args are arguments, which depend on the I/O object type. Some of these arguments may also appear in the I/O object declaration. If specified in both places, the value of the function argument overrides the declared value for that call only. If the value is not specified in either the function argument or the declaration, the default value is used.

See Appendix C for object-specific rules pertaining to *io_in()*.

In this example, the *io_in()* function returns the value of *io_part_detector*:

```
part_detected = io_in(io_part_detector);
```

io_out() Function

When signals need to be sent to a device, an output object is declared and the built-in function `io_out()` is used.

The syntax for `io_out()` is:

```
io_out ( io_object_name, output_value [,args] )
```

The lamp example in Chapter 1 uses `io_out()` to turn the lamp on and off. Here, `nv_lamp_state` is an input network variable bound to the output network variable `nv_switch_state`:

```
io_out(io_lamp_out,  
      (nv_lamp_state != ST_OFF) ? 1 : 0);
```

In the following example, a display LED is attached to pin `IO_0`. The declaration syntax is:

```
#define ON 1  
#define OFF 0  
IO_0 output bit io_display_LED;  
// or  
IO_0 output bit io_display_LED = ON;
```

The second declaration example uses an *initializer*, which tells the system that following a reset, the `io_display_LED` object output value should initially be set to 1. The default initial value is 0.

Now you can control the state of `io_display_LED` by using the `io_out()` function:

```
if (flow_total > 500)  
    io_out(io_display_LED, ON);
```

input_is_new Variable

For all timer/counter input objects, the built-in variable `input_is_new` is set to `TRUE` whenever the `io_in()` call returns an updated value. The data type of the `input_is_new` variable is a short. The frequency with which updates occur depends on the I/O object type.

The following example uses one of the timer/counters. Assume pin `IO_7` is attached to an optical flow meter, which presents a series of pulses that are proportional to the flow of a fluid. The total flow in gallons needs to be determined.

The `pulsecount` input object counts input edges and latches the count approximately every 0.8388608, (specifically, $(2^{23}/10000000)$ seconds). If you were to use the `io_in()` function for this I/O object, you would always read the *current* latched value. If you are summing the total flow, you will need to qualify this operation. Use `input_is_new`, which is set to `TRUE` following an `io_in()` function only if a *new* measurement is made, or in this case, every 0.8388608 seconds.

```
IO_7 input pulsecount io_flow_sensor;  
    // 451 pulses/gallon  
long flow_total, flow_temp;
```

```
.  
. .  
. .  
{  
    flow_temp = io_in(io_flow_sensor);  
    if (input_is_new)  
        flow_total += flow_temp;  
}  
. .  
. .  
.
```

I/O Events

An alternative to using the explicit `io_in()` function is to associate an input object with a predefined event. The two I/O-related predefined events are `io_changes` and `io_update_occurs`. When either event is used, an implied `io_in()` function occurs. These events are used only with input objects and can take a variety of forms. In addition, when evaluated, both the `io_update_occurs` and `io_changes` events perform an implicit `io_in()` function that obtains an input value for the object. A task can access this input value by using the built-in keyword `input_value`. Both events, and the built-in keyword, are further explained in the following sections.

io_changes Event

This event is TRUE when the value read from the input object specified by *io_object_name* changes state. The change can be one of three types:

- any change (an unqualified change)
- a change (in absolute value) by a specified amount (or greater)
- a change to a specified value

The *reference value* is the value read the last time the change event evaluated to TRUE. For the unqualified `io_changes` event, a state change occurs when the current value is different from the reference value.

NOTE: The expression does not need to be a constant. However, a constant expression will be more efficient.

The syntax for this event is:

`io_changes(io_object_name) [by expr | to expr]`

For example, the program `SWITCH.NC` in Chapter 1 uses the `io_changes` event to detect changes in the input bit object `io_switch_in`:

```
when (io_changes(io_switch_in))
```

If you were interested only in when the `io_part_detector` detected a part (a value of `TRUE`, or 1), you could use the following when clause:

```
when (io_changes(io_part_detector) to TRUE)
{
    .
    .
    .
}
```

io_update_occurs Event

The syntax for this event is:

io_update_occurs (*io_object_name*)

The `io_update_occurs` event is `TRUE` when the value read from the input object specified by *io_object_name* has an updated value. The `io_update_occurs` event applies only to timer/counter input objects. Timing for the event depends on the input object type:

ontime and period input event occurs at the end of the time being measured.

pulsecount input event occurs every 0.8388608 seconds, when a new pulse-count value is available.

quadrature input event occurs as soon as at least one count is accumulated.
dualslope event occurs when the conversion is complete.

~~quadrature and totalcount input~~ event occurs as soon as at least one count is accumulated.

pulsecount, quadrature, and totalcount input event occurs if the number of counts measured has changed from the last count.
should read:
pulsecount and quadrature input event occurs if the number of counts measured has changed from the last count.
dualslope event occurs when the conversion is complete.

The `io_changes` event for a timer/counter input device occurs only if the device has a new value, different from the previous value. For the timer/counter devices, the `io_changes` event happens as follows:

ontime and period input event occurs if the measured time has changed from the last time.

pulsecount, quadrature, and totalcount input event occurs if the number of counts measured has changed from the last count.

input_value Variable

The `input_value` built-in variable is a long (`input_value` can be cast in the same manner as any other C variable). For example:

```
when (io_update_occurs(io_dev))
{
    if (input_value > 2) {
        // code
    }
}
```

The lamp example in Chapter 1 sets the value of the network variable `nv_switch_state` based on the value of `input_value` (the switch value):

```
when (io_changes(io_switch_in))
{
    nv_switch_state
        = input_value ? ST_ON : ST_OFF;
}
```

The value of the `input_value` variable depends on the context in which it is used. The following combination of when clauses is valid. Since both events refer to the same I/O object, there is no ambiguity about which object is providing the input.

```
when (io_changes(io_dev) to 4)
when (io_changes(io_dev) to 3)
{
    x = input_value;
}
```

However, the following combination of when clauses is not a valid context for use of `input_value`, since we have no way of knowing which object is providing the input value. If the first when clause evaluated to TRUE, `input_value` would refer to `io_dev2`, but if the second when clause evaluated to TRUE, `input_value` would refer to `io_dev1`.

```
when (io_update_occurs(io_dev2))
when (io_update_occurs(io_dev1))
{
    x = input_value;
}
```


In addition, `input_value` is valid only after an `io_update_occurs` or `io_changes` event. In the following example, using multiple when clauses produces an ambiguous value for `input_value` because the `timer_expires` event does not perform I/O. In such cases, use `io_in()` to retrieve the value.

```
when (timer_expires(t))
when (io_update_occurs(io_dev))
{
    x = input_value;    // use x=io_in(io_dev)
                        // instead of input_value
}
```

Two Methods: Which Should You Use?

You have now read about two different ways to determine whether an input value is new: you can use the `io_update_occurs` event with the `input_value` variable, or you can use the `io_in()` function with the `input_is_new` variable. The following two examples show two ways to accomplish the same goal:

```
IO_5 input pulsecount io_dev;
when (io_update_occurs(io_dev))
{
    if (input_value > 2) {
        // code
    }
}
```

Listing 2-1. `io_update_occurs/input_value`

```
stimer t;
IO_5 input pulsecount io_dev;
when (timer_expires(t))
{
    // code
    if ((io_in(io_dev) > 2) &&
        input_is_new) {
        // code
    }
}
```

Listing 2-2. `io_in()/input_is_new`

Which method you choose depends on the individual case. The I/O event mechanism (that is, use of when clauses, shown in Listing 2-1) is the simpler method, where the scheduler decides when to perform the I/O functions. Use this construct if possible. When you are combining multiple events in a single block of logic, you may need to perform an explicit `io_in()` combined with the `input_is_new` variable, as shown in Listing 2-2.

A Word of Warning

If you combine explicit calls to `io_in()` with when clauses containing I/O events, synchronization problems may result. For example, if a when clause evaluates to TRUE near the end of an I/O sampling period, the `io_in()` might not be executed until the following period, and the value obtained could be misleading:

```
when (io_update_occurs(dev))
{
    // code
    io_in(dev);           // Use input_value instead
                        // of io_in() to retrieve
                        // the value obtained when
                        // the io_update_occurs
                        // event was TRUE
}
```

Relationship between I/O Measurements, Outputs, and Functions.

Direct, Serial, and Parallel I/O Objects

For direct I/O objects, input levels are sampled at the point of the `io_in()` function or the when clause that uses the object.

For serial and parallel I/O objects, input levels are sampled at the point of the `io_in()` function. With a 10 MHz input clock, output levels are set approximately 50 to 100

microseconds after invocation of the `io_out()` function. (This value scales with slower clock speeds.)

Timer/Counter I/O Objects

Values for timer/counter input objects are latched periodically depending on the object type or the object clock. The relationship between when an `io_in()` function or I/O when clause is used and when the data has been latched is usually application dependent. Once a value is latched, that value continues to be returned by `io_in()` until a new value is latched.

The period input and ontime input object types latch a new value on the falling edge of the input signal. (If the `invert` keyword is used, these object types latch the new value on the rising edge of the input signal.) The pulsecount input object latches a new value every 0.8388608 seconds. (See *Input Clock Frequency* and *Timer Accuracy* sections later in this chapter.)

As a general rule, new values written to timer/counter output objects are acted on at the end of the current output signal period. Exceptions to this rule are oneshot output and I/O objects that have been disabled (that is, have a zero control value), all of which take effect upon return from the `io_out()` function.

Also see the *NEURON CHIP Input/Output Timing Specification* Engineering Bulletin (part no. 005-0007-01) for more information.

Output Objects

The following output object types reflect the new `output_value` at the end of the current output signal period:

- frequency output
- triac output
- pulsewidth output

The following I/O object types reflect the new `output_value` upon return from the `io_out()` function:

pulsecount output
oneshot output

All timer/counter output objects respond to a zero
output_value upon return from the io_out() function.

I/O Multiplexing

On both the NEURON 3150 and 3120 CHiPs, input to one of the timer/counter circuits can be multiplexed among pins IO_4 to IO_7 or provide output to IO_0. This timer/counter is referred to as the “multiplexed” timer/counter. A second timer/counter circuit derives input only from IO_4 or provides output to IO_1. This second timer/counter circuit is called the “dedicated” timer/counter. Figure 2-1 shows a signal flow diagram for both the multiplexed and dedicated timer/counter circuits.

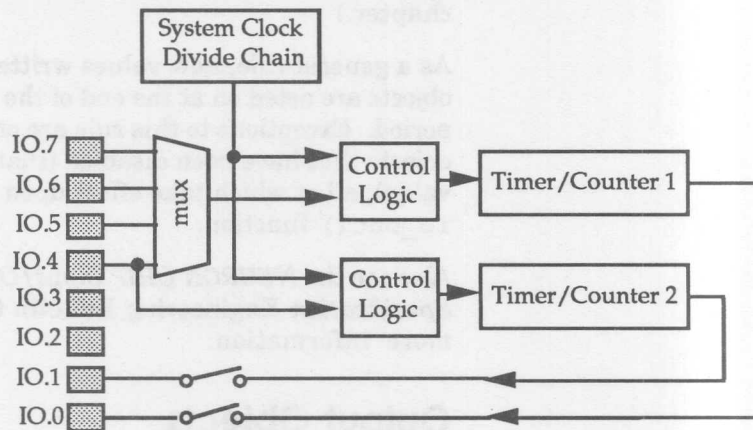


Figure 2-1. Flow Diagram for Timer/Counter Circuits

I/O Functions for Timer/Counter Objects

For multiplexed I/O objects, the last timer/counter I/O object declared in the program is the first to take effect after a reset. To change the selected I/O object, use the io_select()

function to specify which of the multiplexed pins is the owner of the timer/counter circuit. The syntax for `io_select()` is:

`io_select (io_object_name [, clock])`

`io_object_name` is the name for the I/O object, which corresponds to `io_object_name` in the I/O declaration.

`clock` optionally specifies a *clock*, which can be different from or the same as the *clock* specified in the object's declaration, in the range of 0 to 7. If the user does not specify a *clock* value in the call to `io_select()`, the *clock* is set to the value in the object's declaration.

Any timer/counter I/O object that has a clock argument in its declaration syntax can also be reprogrammed to an alternate clock value by use of the `io_set_clock()` function. The syntax for this function is:

`io_set_clock (io_object_name, clock)`

`io_object_name` is the name for the I/O object, which corresponds to `io_object_name` in the I/O declaration.

`clock` specifies a *clock*, which can be any value in the range of 0 to 7 regardless of the *clock* specified in the object's declaration.

When `io_set_clock()` is used on multiplexed objects, the clock is changed regardless of whether the object itself is currently selected.

The following fragment shows several examples of the use of `io_select()` and `io_set_clock()`:

```
IO_1 output pulsecount clock(3) io_pcout;  
IO_5 input period clock(2) io_pdin;  
IO_6 input ontime clock(3) io_ontmin;
```

```
when (reset)  
{  
    io_set_clock(io_pcout, 5);  
    io_select(io_ontmin);  
}  
  
when (io_update_occurs(io_ontmin))  
{  
    .  
    .  
    .  
    io_select(io_pdin, 3);  
}
```

When a new clock is set for an I/O object using `io_select()`, this clock remains in effect until a new value is explicitly set again. The next `io_select()` for the same I/O object will reset the clock to the value specified in the declaration if there is no clock argument in the `io_select()` call.

If an input measurement is attempted using `io_in()` or a when clause on an I/O object that has not been selected with the `io_select()` function, a data value of *overrange* (65,535) is returned, and the `input_is_new` and `io_update_occurs` events remain FALSE.

Following a call to `io_select()` and after a NEURON reset, the first measurement taken for the newly selected I/O object is discarded to clear out any incomplete measurements. The `io_update_occurs` event actually happens when the second measurement is read. Rely on either an `io_update_occurs` event or use the `input_is_new` variable to verify that an actual measurement has been made following an `io_select()`.

The following example shows the use of `io_select()` with the multiplexed timer/counter circuit. For multiplexed I/O objects, the last I/O object declared in the program is the first to take effect after a reset.

```
// I/O Definitions
IO_5 input period mux clock (2) io_pcount_2;
IO_4 input period mux clock (2) io_pcount_1;

static long variable1, variable2;

// The following occurs only when the
// io_pcount_1 is selected

when (io_update_occurs(io_pcount_1))
{
    variable1 = input_value;
    // select next I/O object
    io_select(io_pcount_2);
}

// The following occurs only when the
// io_pcount_2 is selected

when (io_update_occurs(io_pcount_2))
{
    variable2 = input_value;
    // select next I/O object
    io_select(io_pcount_1);
}
```

In the following example, the timer/counter is multiplexed between an ontime measurement on pin IO_5 and a period measurement on pin IO_6. Because the ontime input may cover a large range of values, this example uses a form of "auto-ranging." The clock value switches between 4 and 2 if the input measurement value extends beyond certain values. A variable is used when reselecting the ontime object since its clock may be one of the two values.

```

IO_5 input ontime clock (2) io_slope_1;
IO_6 input period clock (1) io_cycle_a;

unsigned long slopel_raw;
unsigned long cycle_a_value;
int slopel_clock = 2;

// Following reset, the io_cycle_a object is selected
when (io_update_occurs(io_slope_1))
{
    if (input_value > 0x4000 && slopel_clock == 2) {
        // Range down (slower)
        slopel_clock = 4;
        io_set_clock(io_slope_1, 4);
    } else if (input_value < 0x4000 && slopel_clock == 4) {
        // Range up (faster)
        slopel_clock = 2;
        io_set_clock(io_slope_1, 2);
    } else {
        // Save the measured value and select the other object
        slopel_raw = input_value;
        io_select(io_cycle_a);
    }
}

// If auto-ranging has occurred, another measurement will be
// made. Otherwise, the io_cycle_a object will be measured
// next.
}

when (io_update_occurs(io_cycle_a))
{
    io_cycle_a_value = input_value;
    // Now select the io_slope_1 object, using the current
    // clock range
    io_select(io_slope_1, slopel_clock);
}

```

Introduction

The NEURON CHIP parallel I/O object permits bidirectional data transfer at rates of up to 3.3 Mbps. A NEURON CHIP may communicate with another NEURON CHIP, as in a LONTALK application-level router, or with any other microprocessor or microcontroller.

The physical interface to the parallel I/O object is accomplished through the eleven I/O pins of the NEURON CHIP. No other I/O objects of the NEURON CHIP may be used in conjunction with parallel I/O. In addition to the physical interface, a token-passing, handshaking protocol is implemented by the NEURON CHIP firmware as a way to establish synchronization and prevent bus contention.

The NEURON C programming language provides several built-in functions that enable the use of the parallel I/O object without the need for detailed, hardware-level knowledge of the handshaking protocol. See Appendix C for a detailed listing of these functions.

See also the *Parallel I/O Interface to the NEURON CHIP* Engineering Bulletin (part no. 005-0021-01) for more detailed information and examples relating to parallel I/O. For a description of how the parallel I/O object can be used with the Microprocessor Interface Program (MIP) to interface with external host processors, see the *LONWORKS Network Interface Developer's Guide*.

Modes of Operation

For increased design flexibility, the NEURON CHIP provides several modes of operation for the parallel I/O object: master, slave A, and slave B. The different attributes of each mode can be used to tailor the NEURON CHIP for a specific application.

Master Mode

The master mode is the intelligent mode of the parallel I/O object. In this mode, the NEURON CHIP controls the handshaking protocol between itself and the attached processor, which is in the slave A mode. While in the master mode, the NEURON CHIP may be interfaced to another NEURON CHIP (in slave A mode), a microprocessor, or a microcontroller.

Slave A Mode

In the slave A mode, the NEURON CHIP is under control of a master. The master can be another NEURON CHIP (in master mode), a microprocessor, or a microcontroller. In this configuration, only one master and one slave can be connected together.

Slave B Mode

The slave B mode is logically similar in operation to the slave A mode; however, the handshaking process and the data bus control are specifically tailored for use in a microprocessor bus environment. This is useful when interfacing a NEURON CHIP to a microprocessor or microcontroller, or when there is a need for multiple slaves on the same parallel bus (e.g., PC bus interfacing).

Figure 2-2 illustrates the application of the different parallel I/O modes. Although all possible interfacing scenarios are shown, not all can be considered for every application. Certain applications, such as a NEURON CHIP-to-NEURON CHIP connection, have only one solution (master to slave A), while interfacing a host processor to the NEURON CHIP can be accomplished in several ways depending on available hardware and software resources.

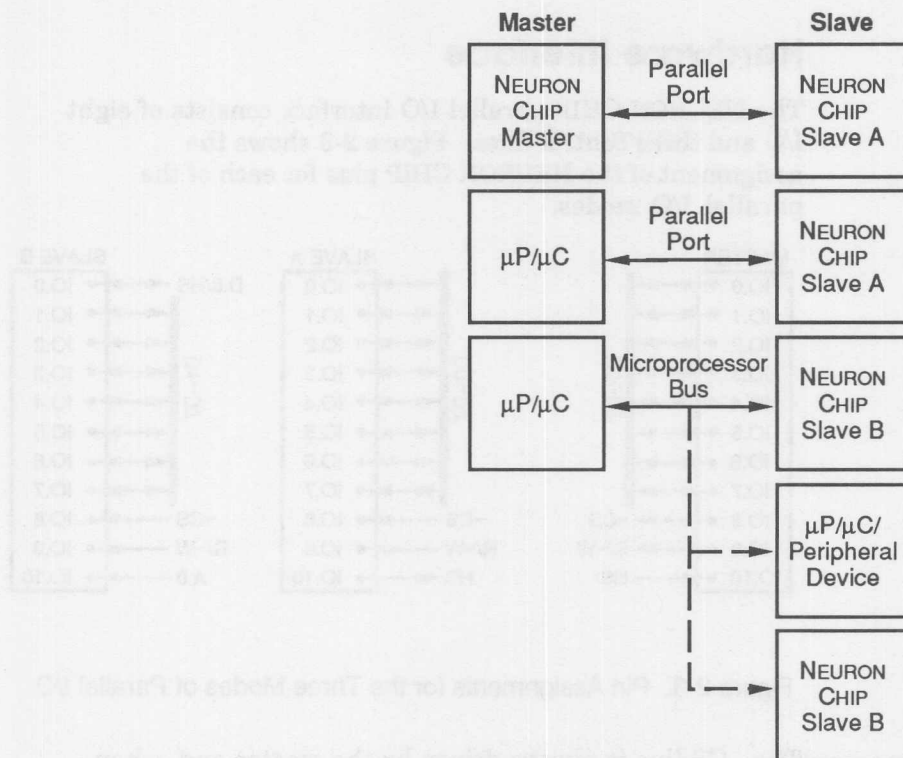


Figure 2-2. Possible Master/Slave Connections for the NEURON CHIP

In a non-NEURON CHIP (host processor) interface, it is assumed that the microprocessor or microcontroller involved has the ability to execute the token passing algorithm dictated by the attached NEURON CHIP. This usually consists of a hardware interface and a software program that duplicates the actions of a NEURON CHIP.

Hardware Interface

The NEURON CHIP parallel I/O interface consists of eight I/O and three control lines. Figure 2-3 shows the assignment of the NEURON CHIP pins for each of the parallel I/O modes.

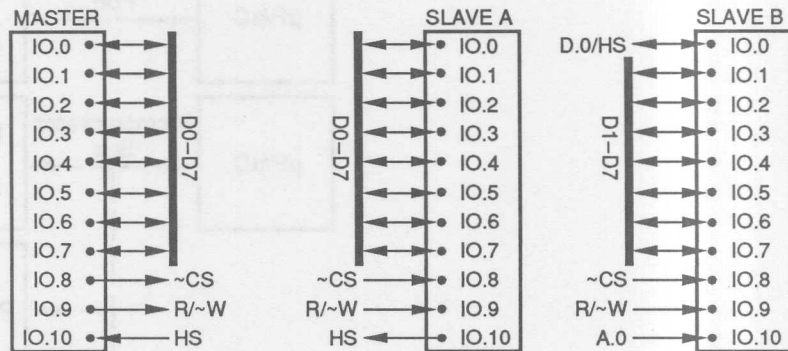


Figure 2-3. Pin Assignments for the Three Modes of Parallel I/O

The ~CS line is always driven by the master and, when active, signifies that a byte transfer operation is currently in progress. A low pulse on this line strobes the data into either the master or the slave.

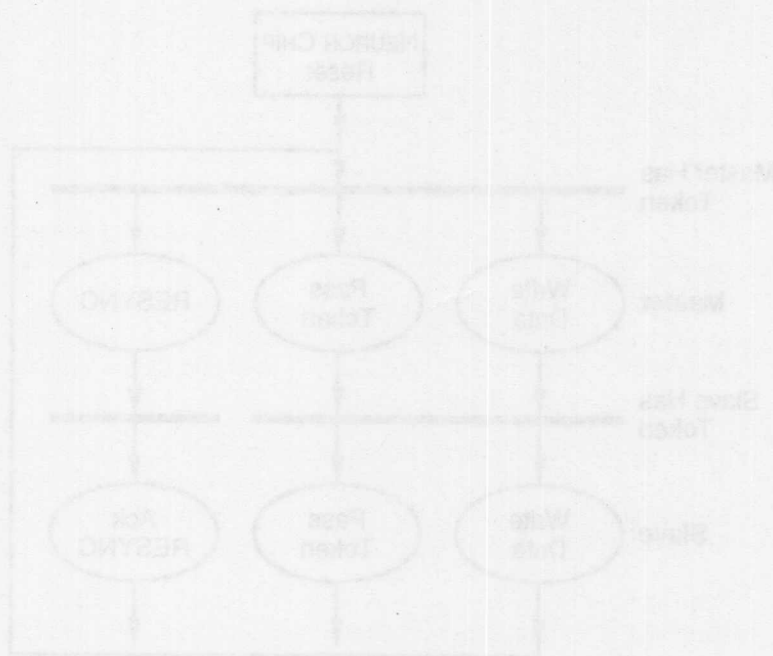
The type of data transfer actually taking place, either a read or a write (with respect to the master), is determined by the level of the R/~W line at the time the ~CS line is pulsed low. The R/~W line is driven by the master.

The HS (handshake) line is always driven by the slave. It informs the master that the slave is busy. In effect, the HS line can be treated as a slave-busy signal. When high, it is the slave's turn to perform an action (read or write command and data); otherwise, it is the master's turn to access the bus.

The A0 pin, driven by the master and only available on the slave B mode, is the address pin that selects between the data register or the control register on the slave containing the

HS bit. The HS bit is the least significant bit of the control register (D0 line). The remaining bits of the control register are unused. The explicit HS polling required by the master is what separates the microprocessor bus-compatible slave B mode from the slave A mode.

It is possible for the master device to come online and poll the HS line before the NEURON CHIP slave has had a chance to set the proper level on this line. To prevent the master from reading invalid data on the HS line, it is recommended that this line be pulled high through a pull-up resistor for a slave A NEURON CHIP. For a slave B NEURON CHIP, the D0 line should be pulled high.



Handshake Protocol

The handshake protocol implemented by the NEURON CHIP firmware permits coexistence of multiple devices on a common bus. At any given time, only one device is given the option of writing to the bus. A virtual write token is passed alternately between the master and the slave on the bus in an infinite, ping-pong fashion. The owner of the token has the option of writing data, or alternatively, passing the token without any data.

Figure 2-4 illustrates the token passing operation between a master and a slave.

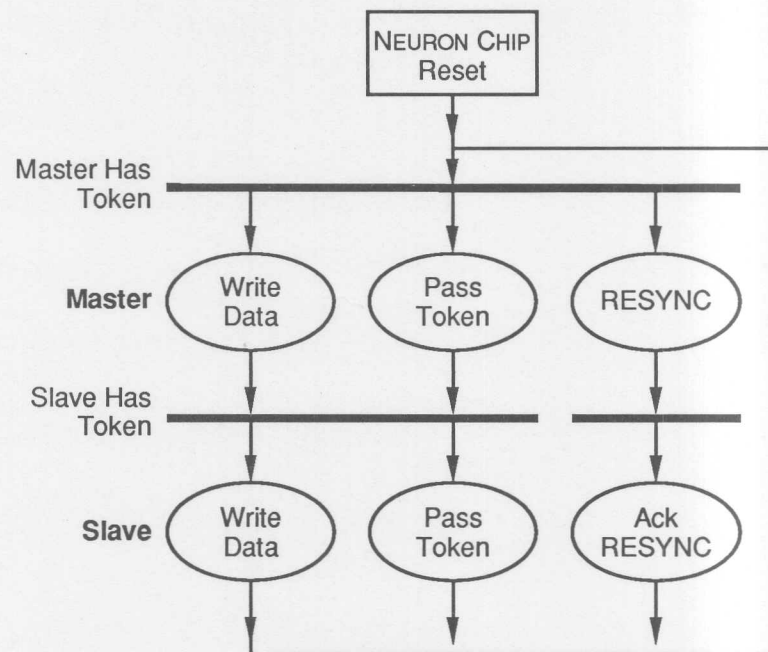


Figure 2-4. Handshake Protocol Sequence Between Master and Slave

Once in possession of the write token, a device may perform one of several operations (as shown in Figure 2-4): write

data, pass token, resynchronize (master only), or acknowledge resynchronization (slave only).

The sequence of events for each of the above operations is the same every time, for either the master or the slave (A or B). However, the degree to which the user is exposed to the underlying token-passing operations is varied depending on the actual device involved. Built-in tools within the NEURON C language allow for straightforward software coding of the NEURON CHIP. This translates to a transparent token-passing protocol, which in turn results in program simplicity and a lower probability of communication errors.

On the other hand, if a NEURON CHIP is interfaced to a non-NEURON processor (host processor), the responsibility of token passing falls in the hands of the attached processor. Although the software program on the NEURON CHIP side is still trivial, the user must now explicitly implement the token-passing protocol on the host processor.

The following describes the operation of data transfer:

<i>R/~W</i>	<i>HS</i>	<i>State</i>	<i>Description</i>
1	0	master_read	Slave CPU has previously written data to its output latch. The slave is waiting for the master to read the data. The HS line is driven high (set) when the output data latch is read by the master CPU.
1	1	slave_write	The slave can write. When the slave CPU writes to the output latch the HS line is reset.

0	0	master_write	The master can write. When the master writes to the slave's input data latch, the HS flag is set.
0	1	slave_read	The master has previously written data to the slave's input latch. The master is waiting for the slave CPU to read the data. When the data is read from the latch, the HS line is reset.

NEURON CHIP-to-NEURON CHIP Interface

The parallel connection of one NEURON CHIP to another is accomplished by assigning one as the master device and the other as a slave A device. The hardware requirements in this case reduce to a direct, one-to-one connection of all eleven I/O pins on both sides.

The following example shows how to use the `io_in_ready` and `io_out_ready` events, in conjunction with the `io_out_request()` function, to handle parallel I/O processing.

Example

For parallel I/O, the `io_in()` or `io_out()` functions should be executed inside tasks associated with `io_in_ready` and `io_out_ready` when clauses. In addition, a call to `io_out()` must be preceded by a call to `io_out_request()`, which sets up the system for an `io_out()` on this parallel object. When the system is ready, the `io_out_ready` event becomes TRUE.

```

IO_0 parallel slave io_bus;

struct parallel_io_interface
{
    unsigned int length; // length of data field
    unsigned int data[DATA_SIZE];
    // DATA_SIZE defined by application
} piofc;

when (io_in_ready(io_bus))
{
    // This event becomes TRUE whenever a
    // message arrives on the parallel bus that
    // must be read. The application must then
    // call io_in() to retrieve the data.
    io_in(io_bus, &piofc);
}

when (io_out_ready(io_bus))
{
    // This event becomes TRUE whenever the
    // parallel bus is in a state where it can be
    // written to and the io_out_request function
    // was previously invoked. The application
    // must then use the io_out function to write
    // the data to the parallel port.
    io_out(io_bus, &piofc);
}

when (...)
{
    // The io_out_request() function is used to
    // request an io_out_ready indication for an
    // I/O object. It is up to the application
    // to buffer the data until the io_out_ready
    // event is TRUE.
    io_out_request(io_bus);
}

```

Master/SlaveA

There are three control lines, R/~W, ~CS, and HS. The master drives the R/~W line, and Chip Select pulse (negative going). The slave drives the Handshake line. The slave NEURON CHIP has holding latches for both data input and output.

Transferring Data from Master to Slave

When transferring data from master to slave (Figure 2-5), the master transfers a byte to the slave's holding register by pulsing ~CS low with the R/~W line stable low. This operation sets the HS latch. When the slave CPU reads the latch, the HS flag is reset. The master CPU can poll the HS after exercising a write to determine when the slave has read the data.

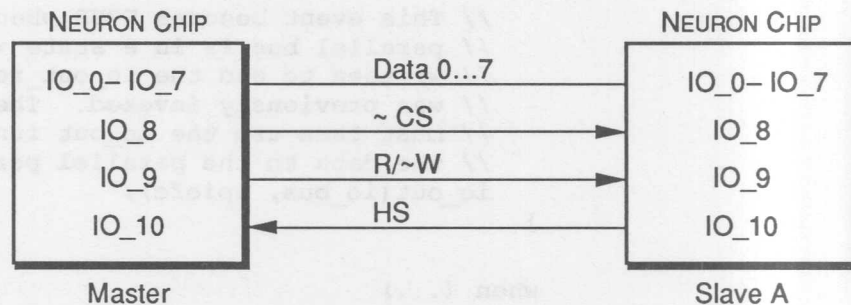


Figure 2-5. Transferring Data from Master to Slave A

Transferring Data from Slave to Master

When transferring data from slave to master, the slave CPU writes data to the output latch (which reset the HS). The master detects that data is available when the HS line goes low (it is assumed that the HS line was idle high, which is accomplished by the protocol described below). The master reads the data by pulsing ~CS low while the R/~W line is

stable high. This operation sets the HS line, providing the trigger for the slave to write the next byte of data to the latch.

Master/Slave B

The NEURON CHIP software is the same for both slave A and slave B; however, the control lines are different.

Slave B mode uses three control lines: R/~W, ~CS, and A0 (the address line). Since only one address line is available, the NEURON CHIP is addressed by either 0 or 1. The master can read or write address 0 (the data address) but only read address 1 (the handshake). The master drives all three control lines. The HS flag is multiplexed on the data bus that is provided to the master on request (A0 high and R/~W high). The slave NEURON CHIP has holding latches for both data input and output, and the handshake.

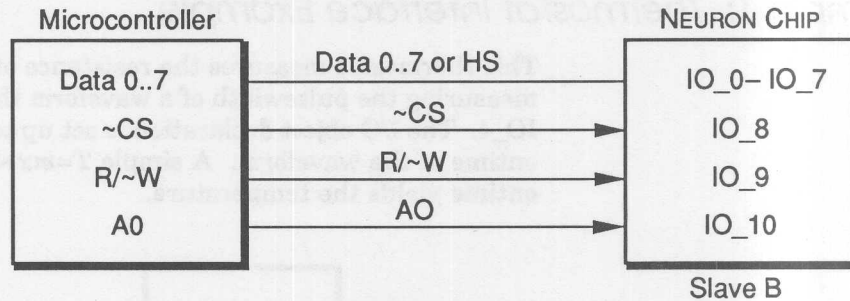


Figure 2-6. Transferring Data from Microprocessor to Slave B

The master transfers a byte to the slave B NEURON CHIP by pulsing ~CS active low with R/~W stable low, A0 stable low, and data stable. The slave B NEURON CHIP latches the data and sets HS latch. When the NEURON CHIP application detects that data is available in the latch, it reads the input latch, which resets the HS latch. The master NEURON CHIP can read the state of the HS latch by pulsing ~CS while R/~W is stable high and A0 is stable high. On indication of the HS being reset, the master can transfer another byte of data.

To transfer a byte of data from the slave B NEURON CHIP to the master, the NEURON CHIP application writes data to its

output latch, which resets the HS flag. The master reads the HS flag to determine when data is available in the slave's output latch. Once there is indication of data, the master pulses \sim CS while R/ \sim W is stable high and A0 is stable low.

Examples

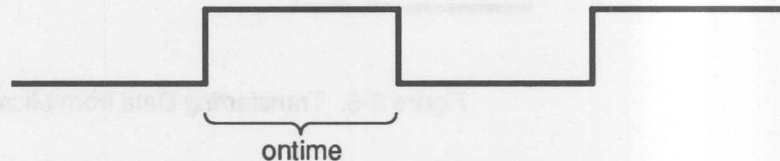
NOTE: These examples work with the LONBUILDER Multifunction I/O Kit if you would like to run the code.

This section presents three complete programs that illustrate NEURON CHIP capabilities and good coding style. The examples are:

- 1 Thermostat interface
- 2 Simple light dimmer interface
- 3 Seven-segment LED display interface

Example 1: Thermostat Interface Example

This thermostat measures the resistance of a thermistor by measuring the pulsewidth of a waveform that is input to pin IO_4. The I/O object declaration is set up to measure the ontime of the waveform. A simple $T=mx+b$ scaling of the ontime yields the temperature.



The example also uses a shaft encoder generating a quadrature input as a dial to select a new temperature setting (see Figure 2-7). The quadrature input object type is used with the `io_update_occurs` event. The input value of the input object represents the change in rotational offset since the last input. Shaft encoders typically generate offsets of 16 to 256 counts per 360 degrees rotation. The `io_update_occurs` event evaluates to TRUE only when a nonzero offset has been measured. In the following application, the task associated with the `io_update_occurs`

when clause is executed only when the quadrature input dial has moved from the previously measured position.

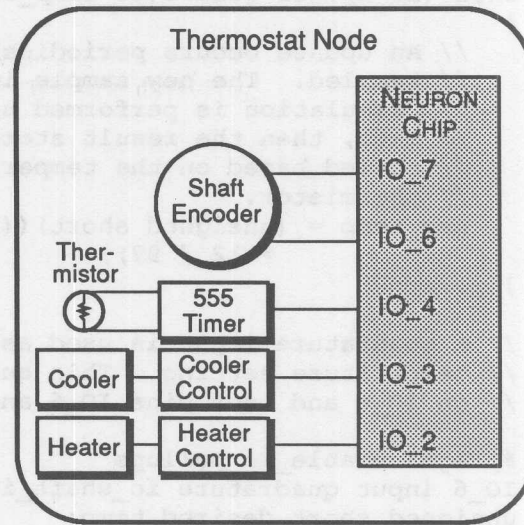


Figure 2-7. Sample Thermostat Node

Note that the `io_changes` event would rarely be used with the quadrature I/O object, since the event would evaluate to TRUE only when a *change* in the measured count occurred. The `io_changes` event would not evaluate to TRUE as long as the input object were moving at a constant rate because the nonzero measurements would be the same. (This example is intended to illustrate use of typical I/O objects. Network variable information has been omitted and is covered in detail in Chapter 3.)

```

IO_4 input ontime ded clock (1) invert io_temp_raw;
unsigned short new_temp;

when (io_update_occurs(io_temp_raw))
{
    // An update occurs periodically as the ontime is
    // sampled. The new sample is placed in 'input_value.'.
    // Calculation is performed using unsigned long (16-bit)
    // math, then the result stored as a short. The input is
    // scaled based on the temperature coefficient of the
    // thermistor.
    new_temp = (unsigned short)((input_value - 30448)
                                * 12 / 97);
}

// A quadrature input is used as a dial to select a new
// temperature setting. This quadrature input is declared
// on IO_6 and uses pins IO_6 and IO_7.

#pragma enable_io_pullups
IO_6 input quadrature io_shaft_in;
unsigned short desired_temp;
const unsigned short desired_temp_max = 84;    // Degrees F
const unsigned short desired_temp_min = 56;    // Degrees F

when (io_update_occurs(io_shaft_in))
{
    // An update occurs for a quadrature I/O object when the
    // accumulated offset is nonzero. The value is placed in
    // 'input_value' by the io_update_occurs event.
    desired_temp += (short)input_value;
    // This assumes no overflow
    desired_temp = min(desired_temp_max, desired_temp);
    desired_temp = max(desired_temp_min, desired_temp);
}

```

```

// A timer is used to decide periodically whether to
// activate heating or cooling. The temperature comparison
// is done only every five minutes to prevent cycling the
// equipment too frequently. There are two digital outputs:
// one for activating the heating equipment, and one for
// activating the cooling equipment.

const int band_size = 2;
// Guardband of +/- 2 degrees around desired temperature
stimer repeating check_heat_or_cool_timer;
// Automatically repeating timer
IO_2 output bit io_heating_on = FALSE;
IO_3 output bit io_cooling_on = FALSE;
typedef enum { OFF, HEATING, COOLING } equip_states;
equip_states equip;

when (timer_expires(check_heat_or_cool_timer))
{
    if (equip == HEATING) {
        if (new_temp > desired_temp) {
            equip = OFF;
            io_out(io_heating_on, FALSE);
        }
    } else if (equip == OFF) {
        if (new_temp < desired_temp - band_size){
            equip = HEATING;
            io_out(io_heating_on, TRUE);
        } else if (new_temp > desired_temp + band_size) {
            equip = COOLING;
            io_out(io_cooling_on, TRUE);
        }
    } else {
        // equip == COOLING
        if (new_temp < desired_temp) {
            equip = OFF;
            io_out(io_cooling_on, FALSE);
        }
    }
}

```



```

// Set up initial conditions. Set new_temp and desired_temp
// to 70, and start the repeating timer, set to expire every
// 300 seconds.
when (reset)
{
    equip = OFF;
    new_temp = desired_temp = 70;
    check_heat_or_cool_timer = 300; // 5 minutes, repeating
}

```


Example 2: Simple Light Dimmer Interface Example

The following example shows NEURON C code for a simple light dimmer. The example uses two I/O objects, a triac control circuit to control the lamp brightness and a quadrature input to select the light level (see Figure 2-8). For the triac output object, a value of 320 is maximum brightness, and a value of 0 is OFF. The initial value on power up is OFF (0).

Note how the `io_update_occurs` event is used in a when clause. An implicit call to `io_in()` occurs when this event is called. The program can then access the measured value through the built-in variable `input_value`.

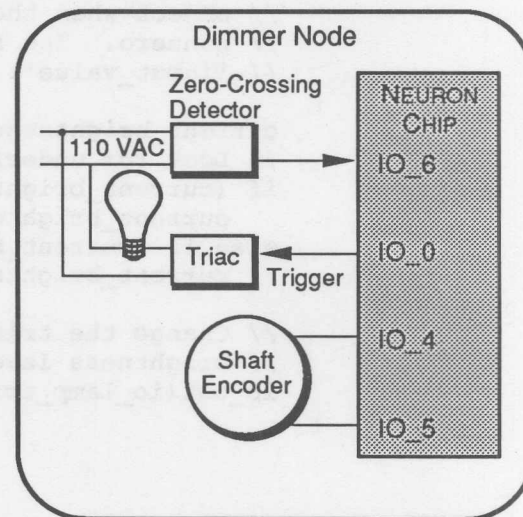


Figure 2-8. Sample Dimmer Node

```

IO_0 output triac sync (IO_6) clock (6)
    io_lamp_triac;
signed long current_brightness;
const unsigned long MAX_BRIGHTNESS = 320;

when (reset)
{
    io_out(io_lamp_triac, 0); //turn lamp off
    current_brightness = 0;
}

// A quadrature input is used as a dial to
// select the light level
IO_4 input quadrature io_shaft_in;
#pragma enable_io_pullups
when (io_update_occurs(io_shaft_in))
    // An update occurs for a quadrature input
    // object when the accumulated offset is
    // nonzero. The sample value is in
    // 'input_value'.
    {
        current_brightness += input_value;
        // Look for underflow or overflow
        if (current_brightness < 0)
            current_brightness = 0;
        else if (current_brightness > MAX_BRIGHTNESS)
            current_brightness = MAX_BRIGHTNESS;

        // Change the triac setting to the desired
        // brightness level.
        io_out(io_lamp_triac, current_brightness);
    }

```

Example 3: Seven-Segment LED Display Interface Example

The following example shows how to connect multi-character displays to the Neurowire port. The display has an 8-bit configuration register and a 24-bit display register. This configuration can be defined as follows:

```
IO_2 output bit io_enable = 1;  
IO_8 neurowire master select(IO_2) io_display;  
unsigned char display_reg[3];  
unsigned char config_reg;  
.  
.  
.  
io_out(io_display, &config_reg, 8);  
io_out(io_display, display_reg, 24);
```

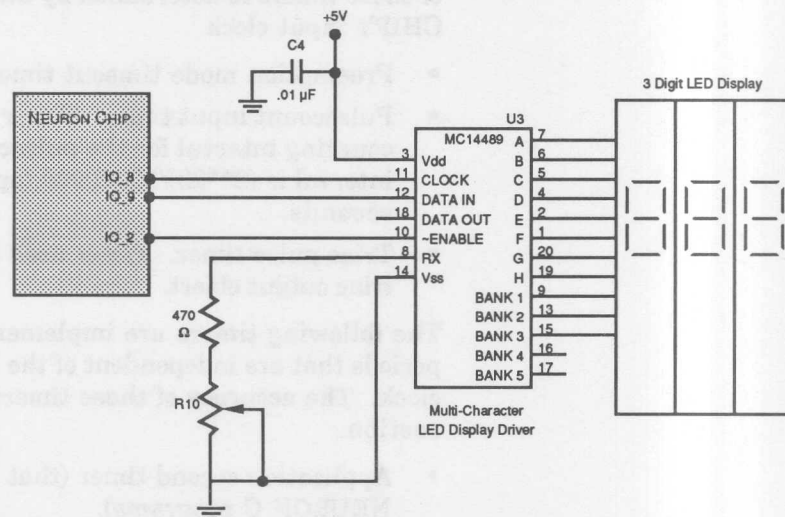


Figure 2-9. Neurowire Connection to a Display

Input Clock Frequency and Timer Accuracy

The NEURON CHIP input clock frequencies are 10 MHz, 5 MHz, 2.5 MHz, 1.25 MHz, and 625 kHz. Certain timers listed below are *fixed timers*; that is, they have the same absolute duration regardless of the input clock selected. However, the slower the input clock, the less accurate the timer. *Scaled timers*, also listed below, scale in proportion to the input clock.

Fixed Timers

In general, timers discussed in this manual are of fixed duration unless noted otherwise. The following timers are implemented in hardware and have periods that are independent of the NEURON CHIP input clock. The accuracy of these timers is determined by the accuracy of the NEURON CHIP's input clock.

- Preemption mode timeout timer.
- Pulsecount input timer. Timer used to determine the counting interval for the pulsecount input object. The interval is $(2^{23})/10000000$ (approximately .8388608) seconds.
- Triac pulse timer. Timer used to generate pulses for the triac output object.

The following timers are implemented in software and have periods that are independent of the NEURON CHIP input clock. The accuracy of these timers is discussed in the next section.

- Application second timer (that is, `stimers` declared in NEURON C programs).
- Application millisecond timer (that is, `mtimers` declared in NEURON C programs).

Scaled Timers and I/O Objects

Timers and I/O objects that scale with the input clock are directly proportional to the input clock. For example, a serial object configured at 2400 bps would actually run at 600 bps given a 2.5 MHz (1/4 speed) oscillator. The following timers scale with the input clock:

- Bitshift timer
- Neurowire timer
- Serial timer
- Watchdog timer
- Configurable EEPROM write timer (for off-chip EEPROM)

Calculating Accuracy for Software Timers

NOTE: When an event is posted by the NEURON CHIP firmware, it becomes visible to the scheduler and to other events (for example, `io_changes`, `nv_update_occurs`).

Accuracy of Millisecond Timers

The following formulas define the range of accuracy for a millisecond timer. Accuracy is expressed as a low and high duration. The “low” duration (L) is the minimum time from when a timer is set to when the system posts an event for the application. The “high” duration (H) is the maximum time from when a timer is set to when an event is posted. L and H are expressed below as a function of E , the expected duration.

The added delay to detect the expiration event is a function of the application and is *not* included in these formulas. For example, an event posted while the application is executing a task associated with a when clause will not be detected until the executing task completes and control of the application returns to the scheduler.

With a Full-Speed Clock (10 MHz)

With a 10 MHz clock, the expected duration of a millisecond timer is:

$$E = .8192 * \text{floor}((D/.82) + 1)$$

where D is the specified duration for the timer. For example, for a timeout of 100 ms, E equals 99.94 ms.

With a 10 MHz clock, the low duration is:

$$L = E - 12 \text{ ms}$$

and the high duration is:

$$H = E + 12 \text{ ms}$$

With Other Clock Speeds

The following formulas allow you to calculate accuracy for millisecond timers when other input clock rates are selected. In these formulas, S depends on input clock speed as follows:

$S =$	Input Clock Rate
1	10 MHz
2	5 MHz
4	2.5 MHz
8	1.25 MHz
16	625 kHz

$$E = .8192 * \text{floor}((\text{floor}(D/S) * S) / .82) + 1)$$

Two factors determine E . The first is that the slower the input clock speed, the less granular the input clock. For example, at 1/16 speed, the millisecond granularity is 16 milliseconds (one clock tick every 16 milliseconds). The second factor is that the hardware generates 819.2 microsecond ticks that the software treats as 820 microsecond ticks. This means that a timer duration is actually .999 times the specified duration.

NOTE: The number "11" is based on a typical worst case scenario. In the absolute worst case, i.e, the maximum number of timers, network variables, addresses, etc., this number can be as high as 32.

For example, with a 2.5 MHz clock, a specified timeout of 99 ms would result in an expected duration of 96.67 ms.

The complete formulas for calculating the low and high durations are:

$$L = E - (11 * S + 1)$$

$$H = E + (11 * S + 1)$$

The high duration with a 2.5 MHz clock and a specified timeout of 99 ms would thus equal 141.67 ms; the low duration is 51.67.

In addition, the high duration may be increased by network management delay (NMD), an additional skew introduced by network management command processing. Normally, this factor is 0. But, if a node were to process a network management command, the upper range for any given timeout could be significantly increased. For example, adding a domain to a node would result in an NMD of anywhere from 300 ms to $(300 + 838 * S)$ ms. In general, network management operations of this type occur infrequently. It is always good practice to take a node offline, if possible, before sending further network management commands.

Repeating Timers

For repeating timers, there is no cumulative drift other than that produced by the difference in D and E . The N th timeout for repeating timers occurs in the range of L_R to H_R , where:

$$E_R = E * N$$

and

$$L_R = E_R - (11 * S + 1)$$

$$H_R = E_R + (11 * S + 1)$$

Note that for repeating timers, intermediate timeout events will be lost if the following is true:

$$|A_R - E_R| \geq E$$

$$E_R - A_R > E$$

where A_R is the actual duration of the repeating timer.

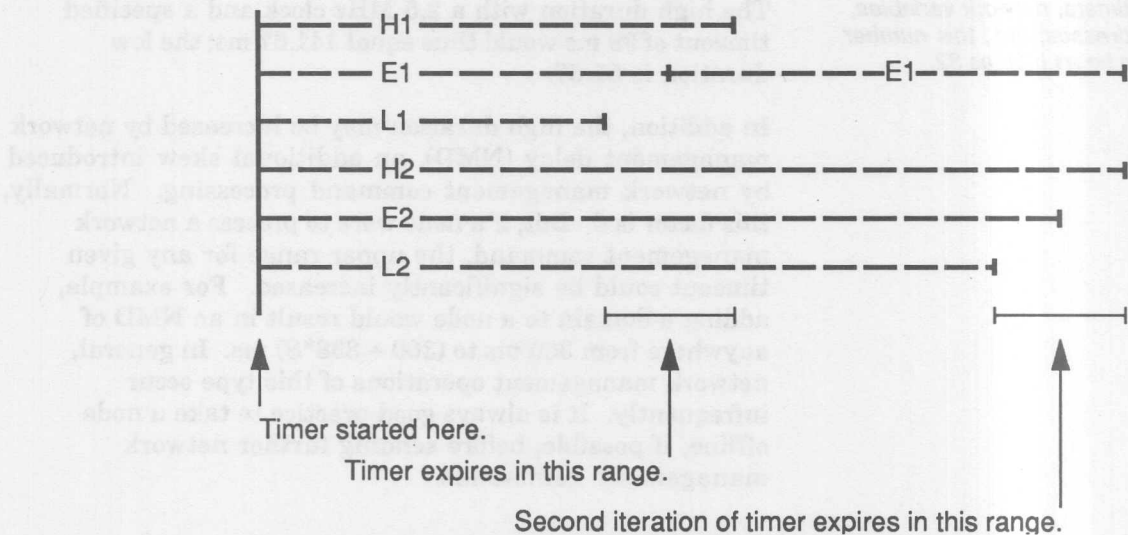


Figure 2-10. Expected, Low, and High Duration of Timeout Events

Accuracy of Second Timers

The second timers rely on the one-second timer, which is based on the millisecond timer mechanism described earlier. A one-second timer of duration D will time out in the range of $D-1$ to D seconds, where "second" is defined as 1001 milliseconds using the millisecond timer duration formulas for L and H .

For example, at 625 kHz, each "second" is of duration 991.23 milliseconds. Thus a 10-second timer would time out in the range of 8.74 to 10.09 seconds.

For repeating one-second timers, the first timeout occurs in the range of $D-1$ to D seconds. Subsequent timeouts occur every D seconds. The fifth timeout of a repeating 10-second timer would occur in the range of 48.39 to 49.74 seconds.

EEPROM Write Timer

The accuracy of the configurable EEPROM write timer degrades with the speed of the input clock. To determine the accuracy of an n millisecond timeout, use the formula:

$$\text{duration} = n * \text{delay}(43)$$

For example, at 625 kHz, a 20 millisecond timeout actually takes 55.2 milliseconds.

Delay Functions

Two functions allow an application to perform timing directly by suspending execution for a given time. These two functions provide a concise way to perform timing in-line:

`delay()`

`scaled_delay()`

The `delay()` function produces a delay of fixed duration that is independent of input clock speed. This function could be used with the wink feature and for I/O debouncing. Its syntax is:

void `delay` (unsigned long count);

count is a value between 1 and 33,333. See Appendix C for the formula used in determining the duration of the delay.

Example:

```
when (io_changes(io_switch))
{
    delay(400); // wait 10 msec for debounce
    .
    .
    .
}
```

The `scaled_delay()` function produces a delay with a duration that scales with input clock speed. Its syntax is:

void `scaled_delay` (unsigned long count);

count is a value between 1 and 33,333. See Appendix C for the formula used in determining the duration of the delay.

3

How Nodes Communicate Using Network Variables

This chapter discusses how nodes communicate with each other using network variables. It includes a detailed discussion of how to declare network variables and how network variables on different nodes are connected to each other. The use of synchronous network variables, the process of polling network variables, and the authentication feature are also described.

Major Topics

Nodes communicate with other nodes through either network variables (which generate implicit messages) or explicit messages. This chapter focuses on network variables, which simplify programming and installation and also reduce program memory requirements. Most programs will use network variables. Explicit messages, if required, are described in Chapter 4. Although this manual discusses the two methods separately, a single program can use both network variables and explicit messages.

This chapter is divided into seven parts:

- *Overview* summarizes the behavior of nodes that are readers and writers of a network variable, as well as how network variables are declared. Also described is how network variables on different nodes are connected to each other.
- *Declaring Network Variables* describes the syntax for declaring network variables, along with related concepts.
- *Connecting Network Variables* describes how network variable readers are connected to network variable writers. (This process was described in general terms in Chapter 1.)
- *Network Variable Events* describes the four scheduling events that are related to network variables:
nv_update_completes, nv_update_fails,
nv_update_occurs, and nv_update_succeeds
- *Synchronous Network Variables* describes the behavior of synchronous network variables.

- *Polling Network Variables* describes how a reader node can poll the writer node for the latest value of a network variable.
- *Authentication* describes how to use authenticated network variables to add a measure of network security. Authentication allows a reader to verify the identity of a writer that attempts to update the reader's value of the network variable. Authentication can also prevent unauthorized configuration of a node.

Overview

As described in Chapter 1, a network variable is an object that may be connected to multiple nodes on a network.

Network variables are first defined within the program that runs on an individual NEURON CHIP. In the Chapter 1 example, the lamp program had one network variable, named `nv_lamp_state` (see Figure 3-1). The switch program also had one network variable, named `nv_switch_state`. The same lamp program is installed on each of the three lamp nodes, and the same switch program is installed on each of the two switch nodes.

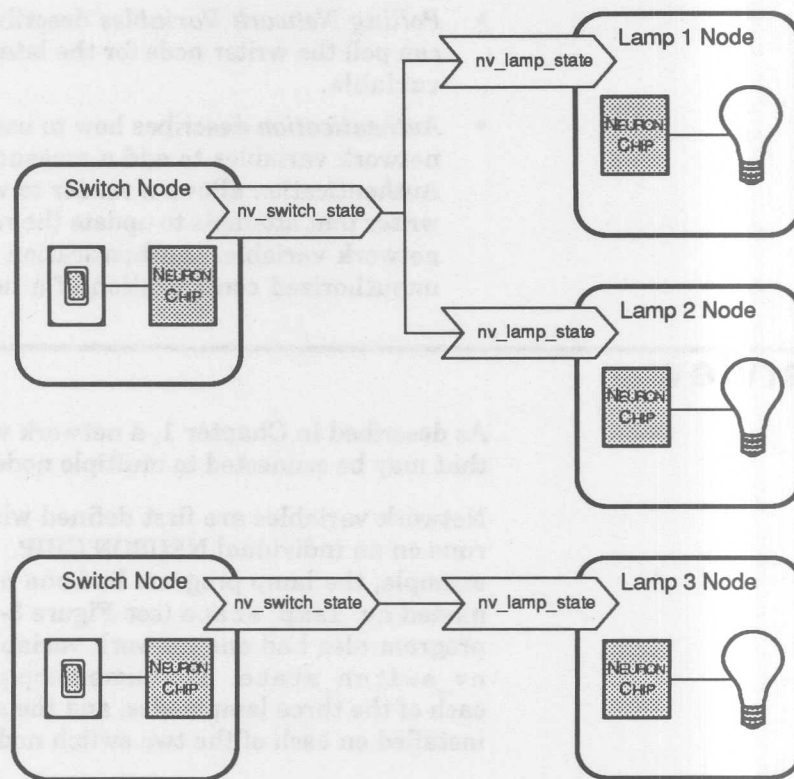


Figure 3-1. Sample Development Network with Five Nodes

The declarations for these two network variables, which appear in different programs, are:

```
network output SNVT_lev_disc nv_switch_state;
```

and

```
network input SNVT_lev_disc nv_lamp_state;
```

These network variables use either of the following network variable syntax structures, discussed in detail below:

NOTE: The brackets around the word "array-bound" do not, in this case, indicate an optional field. They are required and must be keyed in by the programmer.

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [= *initial-value*];

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [*array-bound*] [= *initializer-list*];

The keyword **network** is always followed by the keyword **input** or **output**. The keyword **input** is used for nodes that read the network variable, and the keyword **output** is used for nodes that write to the network variable. In this example, `SNVT_lev_disc` is a Standard Network Variable Type (SNVT). See detailed discussion of SNVTs later in this chapter.

Up to 62 network variables can be declared on a node in a NEURON C program. Up to 4,096 network variables can be declared when using the LONBUILDER Microprocessor Interface Program and an attached host processor. See the *LONBUILDER Host Application Programmer's Guide* for more information.

You can declare an array of network variables using the second form of the syntax shown above. Note that the array can only be single dimension. The *array-bound* must be a constant. Each element of the array is treated as a separate network variable for purposes of events, transmissions on the network, etc. Therefore, each element counts individually towards the maximum number of network variables on a given node. Each element of the array is then a *separately bindable* network variable.

Later you specify connections between network variable outputs and inputs on different nodes. This is discussed in the *Connecting Network Variables* section later in this chapter. The specification of the desired connections is used by a special program within the network management tool, called the binder, to generate the appropriate network addresses. When these addresses are downloaded into the nodes, they ensure that updates sent by writers reach all of the intended readers. A binder is included in the LONBUILDER Developer's Workbench.

In the lamp and switch example, the output network variables in column 1 are connected to the input network variables in column 2.

<i>Output (node / variable_name)</i>	<i>Input (node / variable_name)</i>
switch1/nv_switch_state	lamp1/nv_lamp_state lamp2/nv_lamp_state
switch2/nv_switch_state	lamp3/nv_lamp_state

Behavior of Writer and Reader Nodes

A writer node can change the value of the network variable. The copies of the network variable in all reader nodes are then updated to reflect this change. In general, a reader node only reads from its copy of the network variable. One exception is that a reader node can provide an initial value to the network variable when the variable is declared. Another exception is that a reader node can modify its local copy of a variable in its program. However, in neither case is the new value propagated to any other nodes.

NOTE: This discussion uses the terms writer node and reader node. A writer node is a node that writes to a particular network variable (an output network variable). A reader node is a node that reads a particular network variable (an input network variable). In most cases, a node has both input and output network variables declared in its program, and therefore acts both as a "writer node" and a "reader node", depending on the network variable.

A writer node can also read from its last copy of the network variable, but it will only see the value it wrote last. In other words, two writers of the same network variable cannot change each other's value.

When a network variable declared as output is written to, the NEURON CHIP firmware causes a LONTALK message to be sent to all readers of the variable, informing them of the new value. By default, the message would be sent using the acknowledged (ACKD) service. Note that all readers may not receive updates simultaneously. The network application must be designed to handle update failures and delays.

When Updates Occur

The new value of a network variable received by a reader node does not take effect immediately upon reception and processing of the message. Similarly, assignment of a new value to an output network variable does not cause a message to be sent immediately. Rather, updates occur at the end of a critical section in the application program. A *critical section* is defined as a set of application program statements during which network variable updates are not propagated.

A task is an example of a critical section: once begun, each task runs to completion. When network variable updates are received or requested, they are posted by the scheduler at the end of each critical section. An application can use the `post_events()` function to divide a single task into two or more critical sections. The `post_events()` function forms a boundary at which outgoing network variable updates are sent and incoming network variable updates are processed. See Chapter 5 for further discussion of `post_events()`.

Declaring Network Variables

The complete syntax for declaring a network variable object is one of the following:

NOTE: The brackets around the word "array-bound" do not, in this case, indicate an optional field. They are required and must be keyed in by the programmer.

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [= *initial-value*];

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [*array-bound*] [= *initializer-list*];

Up to 62 network variables (including array elements) may be declared on a node in a NEURON C program. Up to 4,096 network variables can be declared when using the LONBUILDER Microprocessor Interface Program and an attached host processor. See the *LONBUILDER Host Application Programmer's Guide* for more information.

Network Variable Modifiers

The following optional modifiers can be included in the declaration of each network variable:

`sync|synchronized`

specifies that all values assigned to this network variable must be propagated, and in their original order. However, if a synchronous network variable is updated multiple times within a single critical section, only the last value is sent out.

If this keyword is *omitted* from the declaration, the scheduler does not ensure that all assigned values will be propagated. For example, if the network variable is being modified more rapidly than its values can be propagated or more rapidly than its update events can be processed, the scheduler may discard some intermediate data values. However, the most recent value for a network variable will *never* be discarded as long as the node is not reset. See *Synchronous Network Variables* later in this chapter.

`polled`

(used only for output network variables) specifies that the value of the output network variable is to be sent *only* in response to a poll request from a node that reads this network variable. When this keyword is omitted, the value is propagated over the network every time the variable is assigned a value. (However, any reader node can always poll the outputs of

■ Page 3-9 Addition

Add the following network variable modifier after the polled modifier at the top of the page:

`sd_string (<C string constant>)`

is used to set a network variable's self documentation string of up to 1023 bytes. This modifier can only appear once per network variable declaration. The `sd_string` modifier should appear after the `sync` or `polled` modifier(s), if they are used. The ANSI C feature of concatenated string constants is permitted. Each variable's SD string may have a maximum length of 1023 bytes.

writer nodes to which it is connected, whether or not the output is declared as polled.)

Network Variable Classes

Network variables constitute one of the storage classes in NEURON C. They can also be combined with the following storage classes:

NOTE: The NEURON CHIP EEPROM is designed to support at least 10,000 erase/write cycles with no data loss.

`const`

Output network variables declared with `const` are placed in ROM or EEPROM. Input network variables declared with `const` are placed in RAM.

specifies a network variable that cannot be changed by the application program.

`eeprom`

allows the application program to indicate network variables whose values are stored in EEPROM and therefore are preserved across power outages. Note that `eeprom` network variables have a limited capability to accept changes. The initializer for `eeprom` class network variables takes effect when a program is loaded. EEPROM variables are not reinitialized after a reset, but are reinitialized when the application image is reloaded.

`config`

specifies a `const` network variable in EEPROM that can be changed only by another node. This class of network variable is used for application configuration by a network management tool or a network controller.

If no class is specified for a network variable, the network variable is a global variable. Global variables are stored in

the NEURON CHIP's RAM and are not preserved across power outages.

Network Variable Connection Information

connection-info

is an optional field that is used to specify optional attributes of the network variable connections. The following optional fields can be included in the declaration of each network variable:

```
bind_info (  
  [offline]  
  [unackd | unackd_rpt | ackd [(config | nonconfig)]]  
  [authenticated | nonauthenticated [(config | nonconfig)]]  
  [priority | nonpriority [(config | nonconfig)]]  
  [rate_est (const_expr)]  
  [max_rate_est (const_expr)]  
)
```

Each of these fields is described in the *Connection Information* section of Appendix C. The fields can be specified in any order. These connection information assignments can be overridden by a network management tool after a node is installed, unless otherwise specified using the *nonconfig* option.

Network Variable_INITIALIZER

initial-value
or
initializer-list

specifies an initial value (or values) for the network variable. The initial value is loaded with the application image for eeprom and config class network variables. The initial value is loaded on power-up or reset unless the variable is *const*, *eeprom* or *config*. All network variables,

especially input network variables, should be initialized to a reasonable default value. For example:

```
network input SNVT_temp nv_temp = 2692;  
// 22 C, 72 F
```

If a node is reset, this value can be used for subsequent calculations prior to the variable being updated from the network. The default initialization value is 0.

Initializers are not propagated over the network, regardless of whether the network variables are declared input or output.

Network Variable Types

Network variable types serve two purposes. First, typing ensures proper use of the variable in the compilation. Second, typing ensures proper connection of network variables. A network variable can be any of the variable types specified in Chapter 1, except for pointers. The types are:

- [signed] long int
- unsigned long int
- signed char
- [unsigned] char
- [signed] [short] int
- unsigned [short] int
- enums (int type)

- structures and unions of the above types (which *may* contain arrays, but *not* pointers)
- single-dimension arrays of the above types, up to 62 elements

The type can also be a Standard Network Variable Type as described in the next section.

NOTE: The maximum size of a network variable is 31 bytes. In the case of a network variable array, each element is limited to a size of 31 bytes.

When a network variable that is a structure is modified by a network variable writer, the entire structure is updated at the next critical section boundary for all network variable readers, regardless of whether the structure was wholly or partially modified.

Note that you cannot take the address of a network variable with the '&' operator. All updates to a network variable are made by direct assignment.

Network variables may be declared with a single dimension array bound. Each element of the array is then a separately bindable network variable. See the descriptions of the `poll()` function, and the events `nv_array_index`, `nv_update_completes`, `nv_update_fails`, `nv_update_occurs`, and `nv_update_succeeds` in Appendix C for more information.

When an element of a network variable that is an array is modified by a network variable writer, only the modified element is updated at the next critical section.

Standard Network Variable Types (SNVTs)

The LONTALK protocol includes a predefined set of Standard Network Variable Types (SNVTs). A SNVT is used to specify the type in the declaration of a network variable. For example:

```
network input SNVT_temp temp_set_point;
```

By convention, the definition of a SNVT includes units, a range, an increment, and an identification (ID) number between 1 and 250. The ID number may be stored (optionally) in a node's EEPROM whenever a SNVT network variable object is declared. If present, this information can be extracted from the node by a network management tool. A node can thus communicate its interface over the network.

SNVTs should be used for network variables whenever possible because they promote interoperability between nodes.

SNVTs not only ensure compatibility but also ensure that a network management tool can determine the type of a network variable during installation. A network management tool can send a LONTALK network management message to a node to determine the type and optional self-documentation for all network variables declared with SNVTs. See the *SNVT Guide* for the currently defined Standard Network Variable Types.

A SNVT can be used in the definition of a local variable as well as for a network variable. The SNVT definitions are distributed in a binary file called `snvt.typ`. This binary file is read in by the NEURON C compiler each time the compiler is invoked. See the *How to Use SNVTs in LONWORKS Applications Engineering Bulletin* (part no. 005-0002-01) for further information.

Note that applications are free to define their own types for network variables. However, SNVTs are recommended for improved interoperability and simplified installation.

The compiler optionally includes Self-Identification (SI) and Self-Documentation (SD) data in a node's application image. This information identifies certain aspects of each network variable, including its SNVT index, if any. The following three compiler directives can be used to specify and control the generation of SI and SD information:

```
#pragma disable_snvt_si
#pragma enable_sd_nv_names
#pragma set_node_sd_string <C string const>
```

For more detailed information, see the *Compiler Directives* section in Chapter 1.

Examples of Network Variable Declarations

Some sample network variable declarations are:

```
network input SNVT_temp temp_set_point;
network output SNVT_lev_disc primary_heater;
network output int current_temp;
```

Examples of priority network variable declarations are:

```
network output boolean bind_info(priority)
    fire_alarm;
network output boolean bind_info
    (priority(nonconfig)) fire_alarm;
```

An example of declaring a network variable using the unacknowledged service is:

```
network input SNVT_lev_contin
    bind_info(unackd) control_dial;
```

The unacknowledged service can be used for this network variable because the control dial generates numerous messages, and you probably don't need or want to receive an acknowledgment for each one. In addition, it is not critical in this case if a message is not received.

Lamp Example

The lamp example in Chapter 1 could be rewritten to use a config variable. The initial default value of the lamp (nv_lamp_default) would be specified by the network management tool when the node is installed:

```
// lamp.nc - Generic program for a lamp. An input network
// variable controls the lamp's state
#include <snvt_lev.h>
// I/O object declarations
IO_0 output bit io_lamp_control;

// Network variable declarations
network input SNVT_lev_disc nv_lamp_state;

// Configuration data - changeable only by the network
// management tool
network input config SNVT_lev_disc nv_lamp_default = ST_OFF;

// Event-driven code
when (reset)
{
    // Set the lamp to its initial default as specified in
    // configuration
    io_out(io_lamp_control, (nv_lamp_default == ST_OFF)? 0:1);
}

when (nv_update_occurs(nv_lamp_state))
{
    // Use the network variable's value as the new state for
    // the lamp
    io_out(io_lamp_control, (nv_lamp_state == ST_OFF)? 0 : 1);
}
```

Connecting Network Variables

As described in the *LONBUILER User's Guide*, you create the object database before you specify connections for network variables because the binder uses information in the database. The binder selects the appropriate network addressing modes.

The binder first finds all nodes that share common network variables. Then, for each network variable, the binder assigns addresses to all appropriate nodes to ensure that information flows to and from the right places with the minimum number of messages.

Use of the is_bound() Function

The `is_bound()` function indicates whether the specified network variable is connected to any other network variable. Whenever an unconnected network variable is updated, an `nv_update_succeeds` event becomes TRUE even though no update actually occurred (see also the section on *Processing Completion Events for Network Variables* in Chapter 4). Use of this function enables you to avoid executing code that depends on the network variable being connected. The `is_bound()` function can also be used to check whether an input network variable is connected (and thus has a valid value) before you use it. For example:

```
network output SNVT_lev_disc heater_2;

void turn_on_heater_2(void) {
    // turn on secondary heater if one is connected
    if (is_bound(heater_2))
        heater_2 = ST_ON;
}
```

Example of Connecting Multiple Nodes: A Simple Automobile Application

Step 1: Define the problem.

This example describes a simple LONWORKS application used to implement multiplexed wiring in a car. The overall goal of the application is to create a “smart” car that is easy to assemble and easy to upgrade with additional options. The “base” car implemented in this application comes with the following features:

- automatic door locking
- interior light dimming
- “smart” headlights

The application can easily be adapted for either a two- or a four-door car.

For purposes of this introductory discussion, the application is reduced to a small number of nodes (seven) that use a total of five distinct programs. (See Appendix A for a functional definition and the complete program for each node.) Other features that could be added to the application include a digital dashboard, heater/air conditioner, fuel monitoring, security system, bad bulb detectors, and so on.

This example is presented to illustrate connections among multiple nodes and is not intended to represent a complete solution for a particular automobile application.

Figure 3-2 illustrates the hardware components of this application:

- Each car door has a solenoid-activated lock.
- Each car door has a sensor that detects whether the door is closed.
- Each car door has a sensor that detects whether the door is locked.
- The car has one interior light.
- The ignition has an on/off switch controlled by the key.
- The headlights are controlled by a three-position mode switch and an on/off switch.

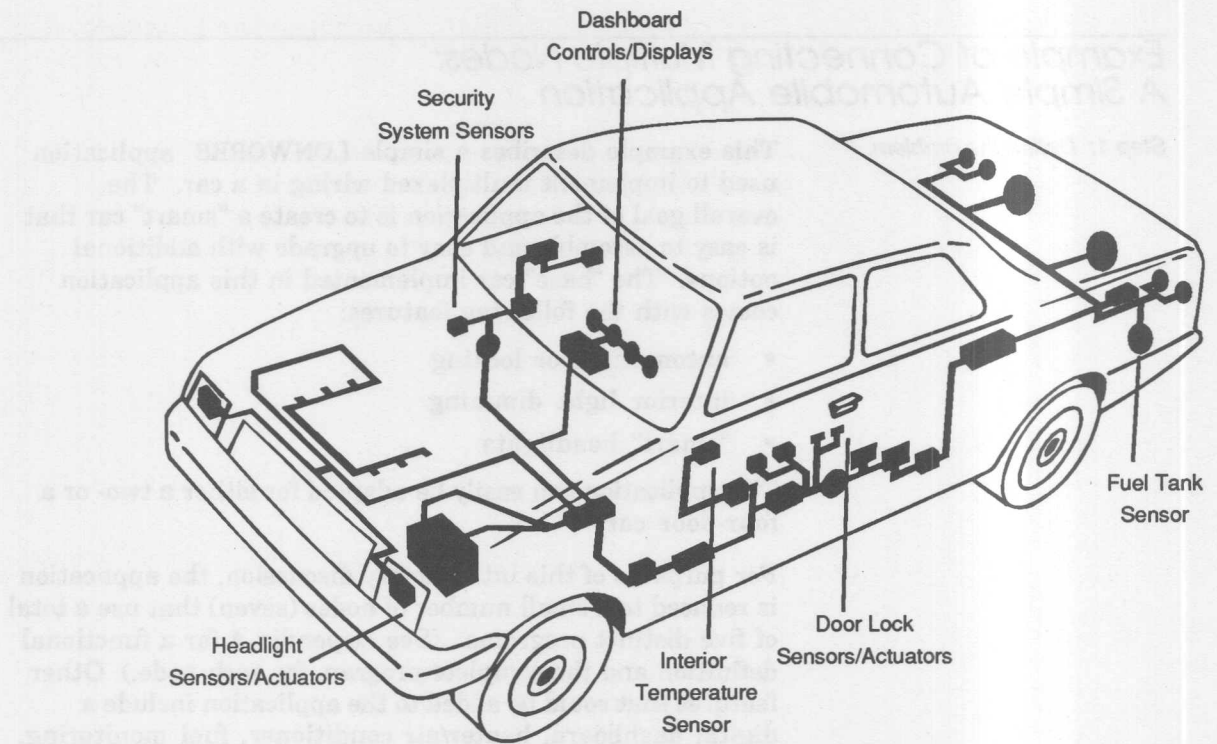


Figure 3-2. Sample Automobile Application

Step 2: Identify nodes and assign their functions.

In a LONWORKS application, each node can be connected to one or more I/O devices. Our example uses seven nodes (see Figure 3-3):

- | | |
|------------------------------|---|
| <i>Left Front Door Node</i> | contains a solenoid-activated lock, a door-open sensor, a control button, and a lock. |
| <i>Right Front Door Node</i> | same as left front door. All doors in the car use the same program. |
| <i>Interior Light Node</i> | contains a manual on/off switch and a light bulb that can be dimmed. |
| <i>Key Node</i> | contains an on/off switch controlled by the key. |
| <i>Headlight Switch Node</i> | contains a manual on/off switch and a three-position switch to indicate headlight mode. |
| <i>Left Headlight Node</i> | contains a headlight that can be dimmed. |
| <i>Right Headlight Node</i> | same as left headlight. Both headlights use the same program. |

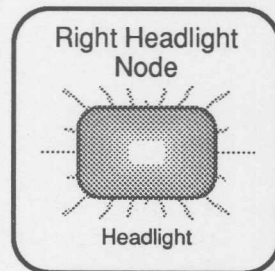
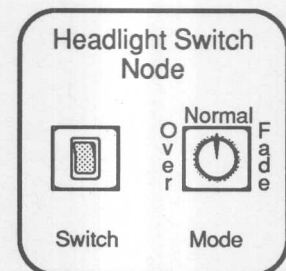
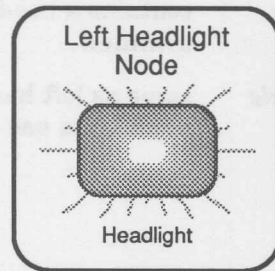
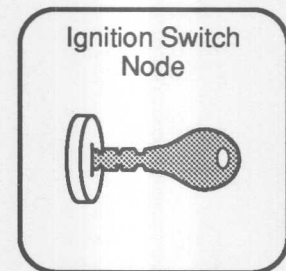
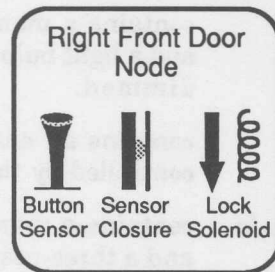
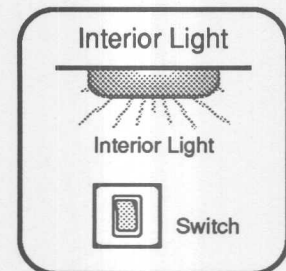
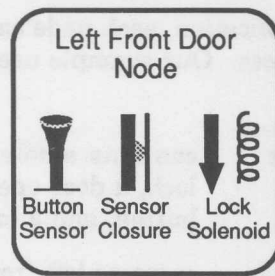


Figure 3-3. Nodes in the Sample Automobile Application

Step 3: Define the external interfaces for each node.

In this application, when either door is opened the interior light goes on. When the left front door is locked, the right front door locks automatically. Figure 3-4 shows the network variables for the two door nodes and the interior light. Output network variables (indicated by arrowheads) are shown connected to the appropriate input network variables (indicated by arrow “tails”.) See Appendix A for a detailed functional definition of each node.

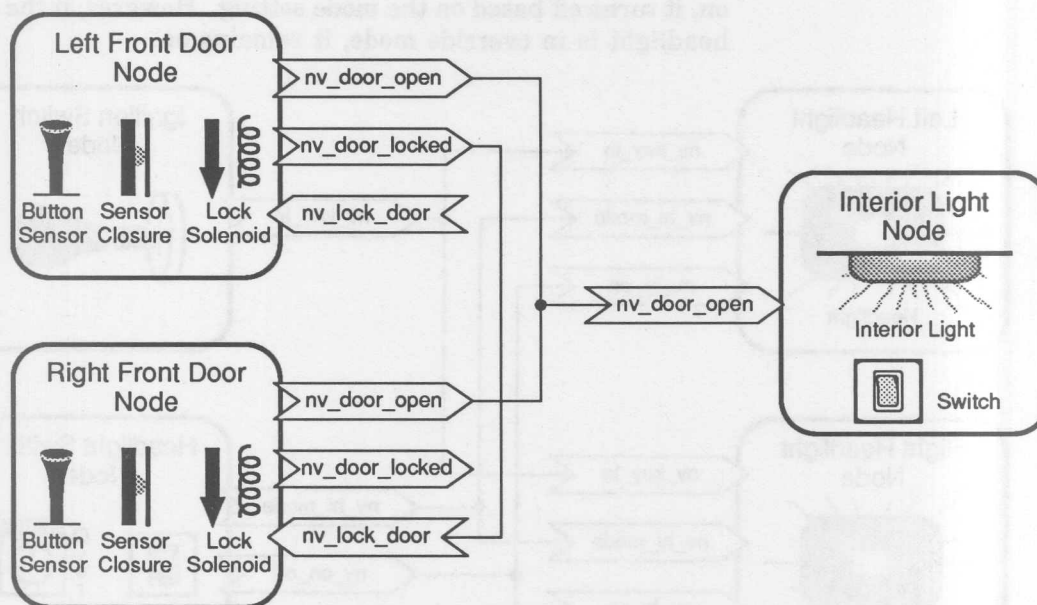


Figure 3-4. Network Variables for Doors and Interior Light

Figure 3-5 shows the network variables for the ignition switch, headlight switch, and the left and right headlights. The headlights use the combined input from the ignition key switch and the headlight switch (input network variables `nv_key_on`, `nv_fade_mode`, and `nv_hl_on` on the headlight nodes). The headlight is initially off. When the ignition key is first turned on, if the headlight on/off switch is on, the headlight turns on at full brightness. If the headlight on/off switch is off, the headlight remains off.

When the ignition key is already in the ON position, if the headlight switch is turned on, the headlight turns on at full brightness. If the headlight switch is turned off, the headlight turns off in one of three modes. In *normal mode*, the light goes off immediately. In *fade mode*, the light slowly fades off. In *override mode*, the light goes off immediately.

When the ignition key is first turned off, if the headlight is on, it turns off based on the mode setting. However, if the headlight is in override mode, it remains on.

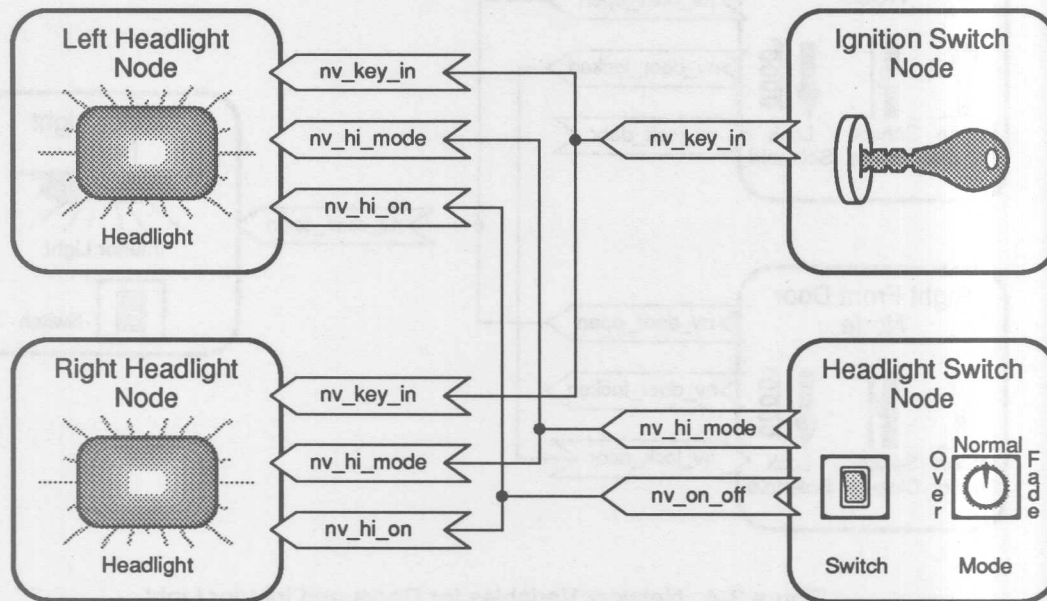


Figure 3-5. Network Variables for Headlight Switch, Key Switch, and Headlights

Figure 3-6 shows nodes and network variable connections for an expanded version of this example.

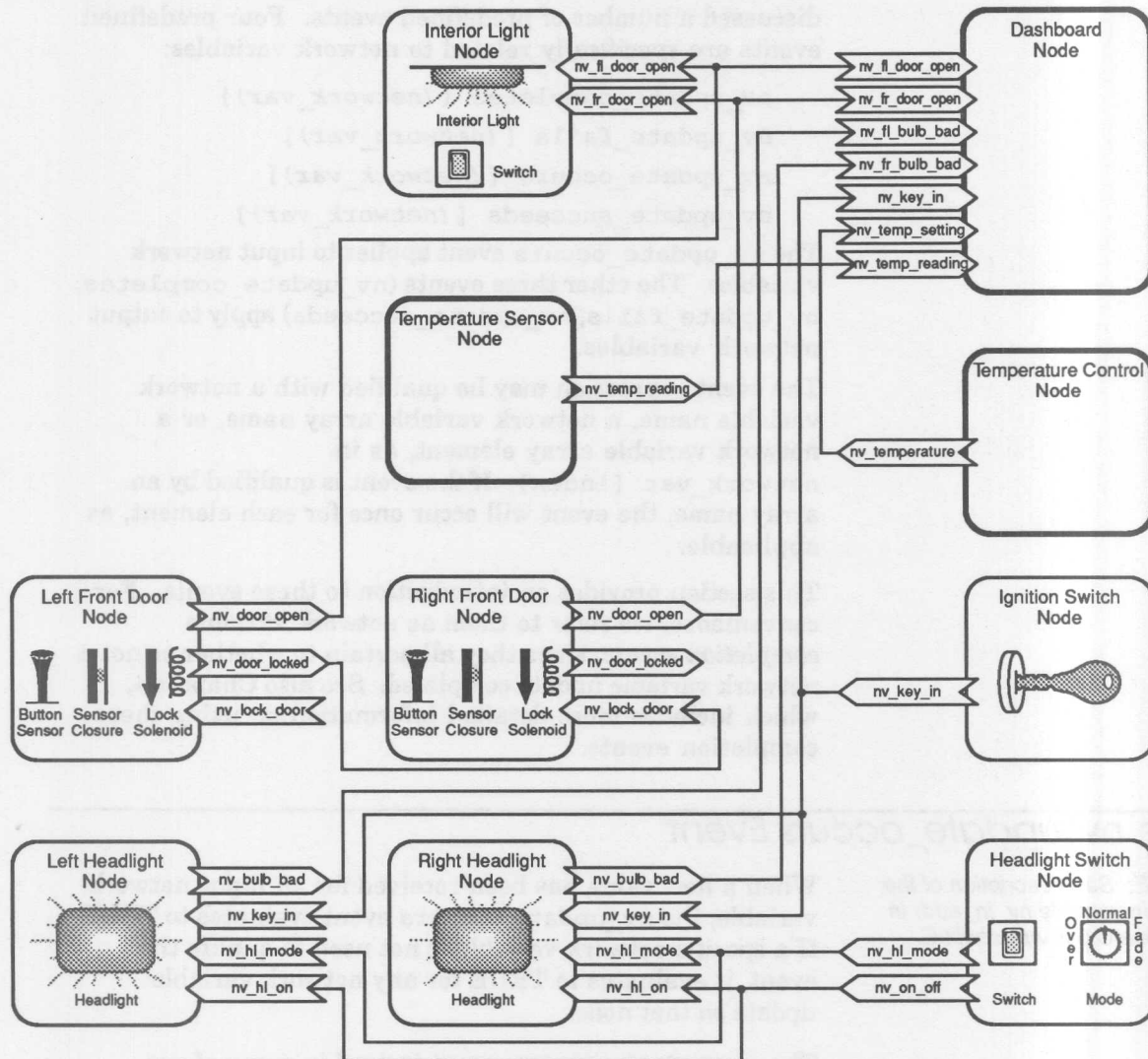


Figure 3-6. Expanded Automobile Application

Network Variable Events

Chapter 2 introduced the event scheduling mechanism and discussed a number of predefined events. Four predefined events are specifically related to network variables:

```
nv_update_completes [(network_var)]
nv_update_fails [(network_var)]
nv_update_occurs [(network_var)]
nv_update_succeeds [(network_var)]
```

The `nv_update_occurs` event applies to input network variables. The other three events (`nv_update_completes`, `nv_update_fails`, `nv_update_succeeds`) apply to output network variables.

The event expression may be qualified with a network variable name, a network variable array name, or a network variable array element, as in `network_var [index]`. If the event is qualified by an array name, the event will occur once for each element, as applicable.

This section provides an introduction to these events. For convenience, we refer to them as network variable *completion events*, since they all pertain to whether or not a network variable update completed. See also Chapter 4, which includes more detailed information on using these completion events.

The `nv_update_occurs` Event

NOTE: See description of the built-in variable `nv_in_addr` in Section C.5 in Appendix C.

When a new value has been received for an input network variable, the `nv_update_occurs` event evaluates to TRUE. If a specific network variable is not used to qualify the event, it evaluates to TRUE for any network variable update on that node.

The `nv_update_occurs` event is used in many of our sample programs. For example, the lamp program uses this event as follows:

```

when (nv_update_occurs(nv_lamp_state))
{
    // Use the network variable's value as the new state for
    // the lamp

    io_out(io_lamp_control, (nv_lamp_state != ST_OFF)? 1 : 0);
}

```

In the following example, when a thermostat node receives a new temperature setpoint, it checks the current temperature and turns the heater on or off if necessary:

```

#include <snvt_lev.h>
network input SNVT_temp temp_set_point;
network output SNVT_lev_disc primary_heater;
network output SNVT_temp current_temp;
when (nv_update_occurs(temp_set_point))
{
    primary_heater = (current_temp <
        temp_set_point) ? ST_ON : ST_OFF;
}

```

A more complex example shows declaring and using enumeration and structure network variable types with `nv_update_occurs`:

```

typedef enum {
    OFF, MASTER, SLAVE, AUTO
} mode_type;
network input mode_type control_mode = OFF;
network output struct {
    int num_control_points;
    long control_points[10];
} ctrl_pt_array;

when (nv_update_occurs(control_mode))
{
    if (control_mode == MASTER) {
        ctrl_pt_array.num_control_points = 2;
        ctrl_pt_array.control_points[0] = 32;
        ctrl_pt_array.control_points[1] = 100;
    }
}

```

The nv_update_succeeds and nv_update_fails Events

When a network variable update or poll fails, the `nv_update_fails` event evaluates to TRUE. If no network variable is specified for the event, it evaluates to TRUE for any network variable update that failed on that node. If multiple network variables are specified, the event can be TRUE once for each network variable update that failed.

Similarly, the `nv_update_succeeds` event evaluates to TRUE whenever an output network variable update has been successfully sent or a polled value has been received.

The switch programs, which all contain output network variables, could use the `nv_update_fails` event. For example:

```
when (nv_update_fails(nv_switch_state))
{
    // take some corrective action
}
```

Here is another example of testing for network update failure and success:

```
boolean heater_node_failed;
network output SNVT_lev_disc nv_heater_1;
when (nv_update_fails(nv_heater_1))
{
    heater_node_failed = TRUE;
    // remember update failure
}
when (nv_update_succeeds(nv_heater_1))
{
    heater_node_failed = FALSE;
    // heater node received update
}
```

The `nv_update_completes` Event

The `nv_update_completes` event evaluates to TRUE whenever an output network variable update or poll either succeeds or fails. An example of testing for network variable update completion is:

```
IO_7 input ontime invert clock(2) io_temperature_sensor;
network output SNVT_temp nv_current_temp;
when (nv_update_completes(nv_current_temp))
{
    ontime_t sensor_value;

    // latest temperature has been sent out on the network send
    // another update
    sensor_value = io_in(io_temperature_sensor);
    nv_current_temp = (sensor_value*221)/642 + 211 + c_to_k;
    // tenths of a degree,C
}
```

If a program checks for `nv_update_completes` or `nv_update_succeeds` for any network variable, then it must check the events for all network variables. The easiest way to do this is to add an unqualified `when (nv_update_completes)` or `when (nv_update_succeeds)` clause after all the qualified `when` clauses for that event. A qualified `when` clause specifies a particular network variable, for example:

```
when (nv_update_completes(some_output_net_var))
```

If an unqualified, nonpriority `when` clause follows qualified `when` clauses for a particular event type, then the program must specify `#pragma scheduler_reset` to ensure that the scheduler evaluates the `when` clauses in the appropriate order.

If a program uses only `nv_update_fails`, then it does not need to check this event for all output network variables.

Sample Program

The following program shows the use of network variable declarations and event processing. Excerpts of this program appear in the preceding paragraphs.

```
// therm.nc: Sample program for a thermostat node that is
// connected to two heater nodes and a temperature setpoint
// node.
#include <io_types.h>
#define c_to_k 2740

// temperature sensor I/O object declaration
IO_7 input ontime invert clock(2) io_temperature_sensor;
IO_2 output bit io_failure_light;
    // LED for heater failure

// Examples of declarations of network variables using SNVTs
network input SNVT_temp nv_set_point;
    //tenths of a degree C+2740, received from setpoint node

network output SNVT_lev_disc nv_heater_1;
    // control heaters (on/off)

network output SNVT_lev_disc nv_heater_2;

network output SNVT_temp nv_current_temp;
    // exported to other nodes

// Function prototype declaration
void heaters_on(boolean state);

// Example of receiving a network variable update event
when (nv_update_occurs(nv_set_point))
{
    heaters_on(nv_current_temp < nv_set_point);
}
```

```

// Example of testing for network variable update completion
when (nv_update_completes(nv_current_temp))
{
    ontime_t sensor_value;

    // latest temperature has been sent out on the network send
    // another update
    sensor_value = io_in(io_temperature_sensor);
    nv_current_temp = (sensor_value*221)/642 + 211 + c_to_k;
    // tenths of a degree,C
}

// Example of testing for network update failure and success
boolean heater_node_failed;
    // true if we cannot communicate with heater

when (nv_update_fails(nv_heater_1))
when (nv_update_fails(nv_heater_2))
{
    heater_node_failed = TRUE;    // remember node failure
    io_out(io_failure_light, 0);  // turn on error indicator
}

when (nv_update_succeeds(nv_heater_1))
when (nv_update_succeeds(nv_heater_2))
{
    heater_node_failed = FALSE;
    // heater node received update
    io_out(io_failure_light, 1);
    // turn off error indicator
}

```

```
// Example of polling a network variable. (See section on
// Polling, later in this chapter)
```

```
when (reset)
{
// when this node starts running, get latest value of
// setpoint
poll(nv_set_point);
io_out(io_failure_light, 1);    // clear error light
heater_node_failed = FALSE;
}
```

```
// Example of using is_bound function
```

```
void heaters_on (boolean state)
{
// control heaters
nv_heater_1 = state ? ST_ON : ST_OFF;
// update primary heater NV

if (is_bound(nv_heater_2))
nv_heater_2 = state ? ST_ON : ST_OFF;
// update secondary heater NV only if it is bound;
}
```

Synchronous Network Variables

It is possible for multiple updates to occur to an input network variable faster than they can be posted by the scheduler. In this case, the scheduler retains only the last value written. Also, when there are multiple updates by different nodes to the same network variable, the *order* of the updates is not preserved. In other words, in cases where there are multiple writers of the same network variable, the last value used by the scheduler is the last value that was received, not necessarily the last update that was sent.

For instances where all updates must be preserved, use a subclass of network variables, the *synchronous* network variable.

Declaring Synchronous Network Variables

Synchronous network variables include the keyword *synchronized* or *sync* in their declaration. For example:

```
network input sync SNVT_temp nv_rel_temp;
```

In the following example, the network variable is declared as synchronous so that all the updates are sent. (If more than one alarm goes off, we want to receive notice of all alarms, not just the most recent one.)

```
// ensure multiple alarms are handled serially
network output synchronized enum {
    none,
    high_temperature,
    high_pressure,
    low_flow,
    control_failure,
} sensor_alarm;
```

Synchronous network variables do not have to be connected to synchronous network variables. (However, the LONBUILDER network manager generates warning messages when the classes are mixed.)

Synchronous vs. Nonsynchronous Network Variables

For most applications, nonsynchronous network variables are adequate and should be used when possible. Many applications need the most *recent* value, not all of the values, for a given network variable. Widespread use of synchronous network variables that are frequently updated could delay processing if the program frequently runs out of buffers (see the *Preemption Mode* section later in this chapter). Depending on the node buffering, channel speed, and congestion of the network, application performance could be adversely affected by extensive use of synchronous network variables.

If a program is required to use relative data values, synchronous network variables are necessary to preserve the intermediate data values. For programs using absolute

data values, nonsynchronous network variables are usually sufficient.

A nonsynchronous output network variable goes out on the network when the next output buffer is available. If the program updates the variable again before that time, only the most recent value goes out. A synchronous output network variable causes the application to wait for an output buffer if none is available. In this case, the scheduler enters preemption mode (see *Preemption Mode* in the next section).

A synchronous input variable always causes an `nv_update_occurs` event when a new value is received and an input buffer is available. For a non-synchronous input network variable, one event is generated even though multiple updates may have been received.

Updating Synchronous Network Variables

Another major difference between synchronous and nonsynchronous network variables is that synchronous network variables are always updated at the end of each critical section. If a buffer is not available, the scheduler waits for one. Nonsynchronous network variables, on the other hand, are updated at the end of critical sections when the scheduler has time to do so. Unlike synchronous network variables, they will not always be updated at the end of the next critical section. As already pointed out, where multiple updates occur, the intermediate values may never be propagated across the network.

Preemption Mode

The scheduler enters *preemption mode* when a synchronous output network variable update occurs and there is no application buffer available. Since the system must send out the synchronous output network variable update, it processes completion events, incoming `msg_arrives` or `nv_update_occurs` events, and response events until an output buffer becomes available. A delay in application processing thus occurs when the system enters preemption mode. The length of the delay depends on how long it takes

for buffer space to become free. This delay depends on network traffic, channel bit rate, and other factors.

Processing Completion Events for Network Variables

For network variables, there are two modes of checking for completion events: partial completion event testing (the default) and comprehensive completion event testing. For message tags (see Chapter 4), only comprehensive completion event testing is available.

Partial Completion Event Testing

Within a given program, for each output network variable, you have two choices on how to process completion events:

- 1 Do not check for any completion events.
- 2 Check for only the failure event (`nv_update_fails`).

For example, within a program containing two network variables:

- Network Variable 1: Program checks for no completion events.
- Network Variable 2: Program checks for failure only.

Comprehensive Completion Event Testing

*NOTE: If you use comprehensive completion event testing features for network variables, all network variable completion code processing must be comprehensive completion event testing. The NEURON C compiler detects use of the comprehensive event feature on a **per-program** basis.*

Comprehensive completion event testing offers the same set of choices for network variable completion events that is available for processing message tag completion events (see Chapter 4). Within a given program, for each output network variable, you have three choices of how to process completion events:

- 1 Do not check for any completion events.

- 2 Check for the failure and the success events
(nv_update_fails, nv_update_succeeds).
- 3 Check for the update completion event
(nv_update_completes).

For example, within a program containing three network variables:

- Network Variable 1: Program checks for no completion events
- Network Variable 2: Program checks for failure and success.
- Network Variable 3: Program checks for update completion.

Tradeoffs

Using comprehensive completion event testing for processing network variable completion events within a program requires more code space and is less efficient than use of partial completion event testing. Note that if you choose a comprehensive completion event testing feature, such as checking `nv_update_completes`, you are limited to comprehensive completion event testing features for whichever network variable's events in which you are interested. For example, within a comprehensive completion event testing program, you cannot simply check for `nv_update_fails`, because that feature applies only to partial completion event testing.

Polling Network Variables

As described earlier in this chapter, a network variable update is initiated when a writer node assigns a value to a network variable. In this usual case, the network variable update is initiated by a writer node.

A reader node can also request that the writer node send its latest value for a network variable. The term *polling* refers to this process in which a network variable update is initiated by a reader node. A node's program may poll any input network variables at any time, including initial

power up and when transitioning from offline to online. Polling on initial power-up can cause network congestion if many nodes are powered-up at the same time, and they all do power-up polling. The reader node makes its request through the `poll` function. The syntax is:

`poll ([network_var]);`

network_var is an input network variable identifier

If no network variable is specified, all input network variables for the node are polled. Note that no explicit “polled” declaration is required.

The *network_var* identifier may also be a network variable array identifier, or an element of a network variable array, as in `network_var [index]`. If a network variable array name is used without an index, all elements of the array are polled.

The new value resulting from the poll may not be immediately available after the `poll()` function call. Use a qualified `nv_update_occurs` event in a `when` clause or some other conditional statement to obtain the new, polled value.

Example

```
when (timer_expires(t))
{
    poll(nv_cooling_mode);
    .
    .
    .
}
when (nv_update_occurs(nv_cooling_mode))
{
    .
    .
    .
}
```

The lamp program presented in Chapter 1 includes a poll of the input network variable `nv_lamp_state` after a reset event. The node obtains the most recent value of `nv_lamp_state`, and then use that value after reset.

```
// lamp.nc - Generic program for a lamp. An input network
// variable controls the lamp's state
#include <snvt_lev.h>
// I/O declarations
IO_0 output bit io_lamp_control = 0;

// Network variable declarations
network input SNVT_lev_disc nv_lamp_state;

// Event-driven code
when (reset)
{
    // Request last value from any switch attached
    poll(nv_lamp_state);
}

when(nv_update_occurs(nv_lamp_state))
{
    // Use the network variable's value as the new state for
    // the lamp
    io_out(io_lamp_control, (nv_lamp_state != ST_OFF)? 1:0);
}
```

Declaring a Network Variable as Polled

Poll requests are initiated by reader nodes. A writer node may assign values to its network variable(s) frequently, but the reader node may want to receive these updates only at specified times. The output network variable in this case should be *declared* as polled:

network output polled type netvar;

In this special case, the output network variable's value is never propagated as a result of its value changing. Instead, the output network variable's value is sent *only* in response to a poll request from a reader node.

Example

The lamp/switch example in Chapter 1 could also be written to use explicit polling of the switch network variable. Complete programs illustrating polling are shown below.

```
// lamp.nc: Generic program for a lamp that polls the switch
#include <snvt_lev.h>
// I/O declarations
IO_0 output bit io_lamp_control = 0;

// Network variable declarations
network input SNVT_lev_disc nv_lamp_state;

// Timers
mtimer poll_timer;

// Event-driven code
when (reset)          // Go get the lamp setting ...
when (timer_expires(poll_timer))
    // Time to poll again...
{
    poll(nv_lamp_state);
}
```

```

when (nv_update_occurs(nv_lamp_state))
{
    // Use the network variable's value as the new state for
    // the lamp
    io_out(io_lamp_control, (nv_lamp_state == ST_OFF)? 0 : 1);

    poll_timer = 500; // Wait 500 msec before polling again
}

```

Listing 3-1. Lamp Program Using Polling

```

// switchp.nc - Generic program for a polled switch. Set a
// polled output network variable to the switch's current
// state
#include <snvt_lev.h>
// I/O Declarations
IO_4 input bit io_switch_in;

// Network variable declarations
network output polled SNVT_lev_disc nv_switch_state;

// Event-driven code
when (reset)
{
    nv_switch_state=(io_in(io_switch_in)==0)? ST_OFF : ST_ON;
    io_change_init(io_switch_in);
}

when (io_changes(io_switch_in))
{
    // Just assign the new switch value (input_value) to the
    // network variable (nv_switch_state). In this case, no
    // message is sent until a poll request is received from
    // a reader node

    nv_switch_state = (input_value == 0) ? ST_OFF : ST_ON;
}

```

Listing 3-2. Switch Program Using Polling

Monitoring Network Variables

A network monitor is a LONWORKS node that must monitor many other nodes. The nodes being monitored are typically identical. For example, an alarm display node may monitor many alarm sensor nodes. The sensor nodes may all have a network variable output declared as SNVT_lev_disc output, and the monitor node may have a network variable input, also declared as a SNVT_lev_disc.

Typically, the monitor node waits for a change to its input network variable. When a change occurs, it must identify which node originated the change. The method of determining the source of a change depends on the method used to connect the sensor outputs to the monitor input. Following are a few options for the network monitor node; in the examples, the sensor nodes all have a single SNVT_lev_disc output network variable that must be monitored by the network monitor node:

- Declare the network variable input as an array, and connect each element of the array to a different sensor. Wait for an nv_update_occurs event for the entire array, then use the nv_array_index built-in variable to determine which node originated the change. For example:

```
network input SNVT_lev_disc nv_alarm_array[50];

SNVT_lev_disc alarm_value;
unsigned int alarm_node;

when (nv_update_occurs(nv_alarm_array))
{
    alarm_node = nv_array_index;
    alarm_value = nv_alarm_array[alarm_node];

    // Process alarm_node and alarm_value ...
}
```

This method is appropriate when the number of nodes to be monitored does not exceed the network variable limits of the monitoring node: 62 for a NEURON CHIP-based node; 4,096 for a MIP-based node.

- Declare the network variable input as a single input on the monitor node, and declare the network variable outputs as polled outputs on the sensor nodes. Create a single connection with all the sensor outputs and the monitor input. Explicitly poll each of the sensors using explicit addressing and explicit messages as described in the next chapter. Since the nodes are explicitly polled, the monitor node always knows the source of a network variable update.

This method is appropriate for any number of nodes, as long as the delays introduced by the polling loop are acceptable for the application.

- Declare the network variable input as a single input and create a single connection with all the sensor outputs and the monitor input. Wait for an `nv_update_occurs` event for the network variable input, then use the `nv_in_addr` built-in variable to determine which node originated the change. Use LONTALK network management messages to identify the originating node. For example, the read memory network management message can be used to read the location string of the sensor node. Following is an example of the code on the network monitor node:

```
network input SNVT_lev_disc nv_alarm_in;

SNVT_lev_disc alarm_value;
nv_in_addr_t  alarm_node_addr;

when (nv_update_occurs(nv_alarm_in))
{
    alarm_node_addr = nv_in_addr;
    alarm_value     = nv_alarm_in;

    // Process alarm_node_addr and alarm_value

    // Use network management messages to get
    // information from the node at
    // alarm_node_addr
}
```

This method is appropriate for any number of nodes.

See Chapter 4 for a description of how to send explicit messages. Appendix C describes the contents of the `nv_in_addr` built-in variable. See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the LONTALK network management messages. See the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for examples of sending network management messages from NEURON C programs.

Authentication

Authentication is a special form of an acknowledged service between one writer node and from 1 to 63 reader nodes.

Authentication is used to verify the identity of the writer node. This type of service is useful, for example, if a node containing an electronic lock receives a message to open the lock. By using authentication, the electronic lock node can verify that the “open” message comes from the owner, not from someone attempting to break into the system.

Authentication doubles the number of messages per transaction. An acknowledged message normally requires two messages, an update and an acknowledgement. An authenticated message requires four messages, as shown in Figure 3-7. This may affect system response time and capacity.

The following paragraphs describe how to set up nodes to use authentication and how authentication works.

Setting Up Nodes to Use Authentication

You must perform these two steps before a program can use authenticated network variables or send authenticated messages. Each of these steps is described in more detail below.

- 1 Declare the network variable as authenticated. For explicit messages to be authenticated, specify TRUE in the authenticated field of the msg_out object.
- 2 Specify the authentication key to be used for this node using a network management tool. The LONBUILDER Developer's Workbench is used to install a key during development.

Declaring Authenticated Variables and Messages

NOTE: The keyword *authenticated* can be abbreviated as *auth*. Likewise, the keyword *nonauthenticated* can be abbreviated as *nonauth*.

For network variables, the authenticated (or auth) keyword is included as part of the connection information. The syntax is:

```
bind_info ( [authenticated | nonauthenticated ]  
           [(config | nonconfig)] )
```

Including the additional keyword *config* in the declaration allows the network management tool to change the authentication status of this network variable after a node has been installed. Setting *nonconfig* prevents the authentication status from ever being changed for this network variable.

For example:

```
network output boolean  
    bind_info(auth(nonconfig)) nv_safe_lock;
```

With this declaration, authentication can never be turned off for updates of the *nv_safe_lock* network variable, because the declaration includes the *nonconfig* keyword.

Specifying the Authentication Key

The second step is to specify this node's authentication key. In the LONBUILDER Developer's Workbench, this key is specified, in hexadecimal, in the Node Create screen of the Navigator under the "App Node/Node Specs" heading. If a node belongs to two domains, then two keys are specified, one for each domain. All nodes that read or write a given authenticated network variable must have the same authentication key. This 48-bit authentication key is used in a special way for authentication, as described below.

The key itself is transmitted to the node only during the initial configuration. All subsequent changes to the key do not involve sending it over the network. The network management tool can modify a node's key over the network, in a secure fashion, with a network management command.

How Authentication Works

The following sequence describes an example of authentication. Figure 3-7 illustrates the process.

- 1** Node A sends an update to a network variable declared as authenticated on Node B using the acknowledged service. If Node A does not receive the challenge, it sends a retry of the initial update.
- 2** Node B generates a 64-bit random number and returns to Node A a challenge packet that includes the 64-bit random number. Node B then uses the encryption algorithm (built in to the NEURON CHIP firmware) to compute a transformation on that random number using its 48-bit authentication key and the message data. The transformation is stored in Node B.
- 3** Node A then also uses the encryption algorithm (built in to the NEURON CHIP firmware) to compute a transformation on the random number (returned to it by Node B) using its 48-bit authentication key and the message data. Node A then sends this computed transformation to Node B.

- 4 Node B compares its computed transformation with the number it receives from Node A. If the two numbers match, the identity of the sender is verified, and Node B can perform the requested action and send its acknowledgement to Node A. If the two numbers do not match, Node B does not perform the requested action and an error is logged in the error table.

If the acknowledgment is lost and Node A tries to send the same message again, Node B remembers that the authentication was successfully completed and acknowledges it again.

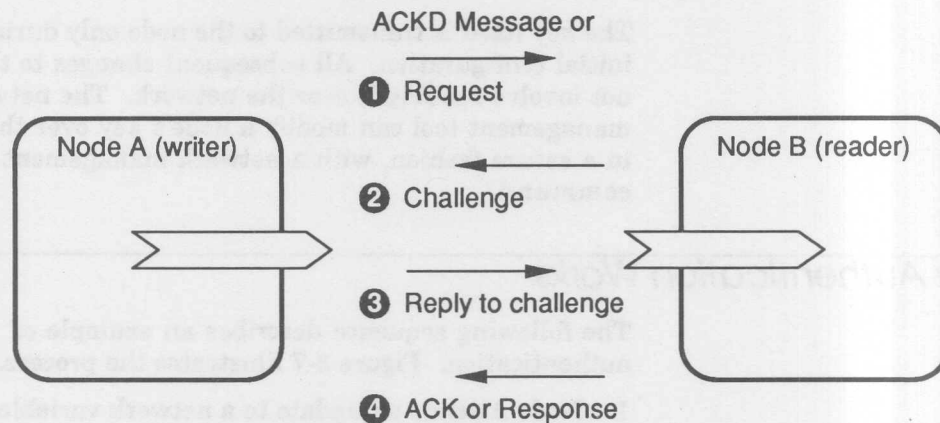


Figure 3-7. Authentication Process

If Node A attempts to update an output network variable connected to multiple readers, each receiver node generates a different 64-bit random number and sends it in a challenge packet to Node A. Node A must then transform each of these numbers and send a reply to each receiver node.

The principal strength of authentication is that it cannot be defeated by simple record and playback of commands that implement the desired functions (for example, unlocking the lock). Authentication does not require that the specific messages and commands be secret, since they are sent

unencrypted over the network, and anyone who is determined can see what those messages are.

It is good practice to connect a node directly to a network management tool when installing its authentication key the first time. This prevents the key from being sent over the network where it might be detected by an intruder. Once a node has its authentication key, a network management tool can modify the key, over the network, by sending an increment to be added to the existing key.

4

How Nodes Communicate Using Explicit Messages

This chapter describes the use of explicit messages, which can be used in place of or in addition to network variables. The request/response mechanism, a special use of explicit messages, is also described. Other topics covered here include preemption mode, asynchronous and direct event processing, the use of completion events with messages and with network variables, and authentication for messages.

Explicit Messages vs. Network Variables

As described in Chapter 3, nodes can communicate with other nodes through network variables (which generate implicit messages) or through explicit messages. Depending on the individual case, there may be advantages and disadvantages to the use of each of these methods. A few of these pros and cons are listed in the following paragraphs. They are described in more detail later in this chapter.

With messages, it is easy to have a variable size data field. The same message code could have one byte of data in one instance and 25 bytes of data in another instance. The size of the data field is constant for a given network variable. With messages, you can control buffer allocation in a program.

Explicit messages provide a request/response mechanism, which enables an application on one node to cause an application on another node to respond to it. The request/response mechanism is similar to a network variable poll, but it provides additional functionality. When a network variable is polled, the application scheduler on the polled node provides the most recent value for that network variable, without intervention of (or knowledge by) the application program. In contrast, when a message is sent with the request service, the application program on the remote node actually takes some action as a result of receiving the message, and then provides a new value for its response. The request/response mechanism also provides the basis for remote procedure calls, since it provides a way for an application on one node to invoke an action on another node.

Explicit messages do not promote interoperability as do network variables because the format of the data portion of an explicit message is application dependent.

Explicit messages use less EEPROM table space than network variables. Explicit messages always use more code space than network variables. In addition, explicit messages are a more complex method of conveying

information from one node to another. The programmer must explicitly build, send, and receive explicit messages. Message attributes such as service type, authentication, and priority are defined at compile time and are not configurable by a network management tool after node installation (however, these attributes can be set on a message-by-message basis).

Layers of NEURON CHIP Software

When you use network variables in a program, the actual building and sending of messages takes place behind the scenes. As shown in Figure 4-1, three layers of software are involved: the application layer, which includes the scheduler, the network layer and the Media Access Control (MAC) layer. Each of these layers of software corresponds to one or more layers of the LONTALK protocol and is handled by a separate processor on the NEURON CHIP.

Note that only one of these layers, the application layer, can be programmed. Your program also has access to some of the information provided by the network layer through the services of the scheduler, as described later in this chapter.

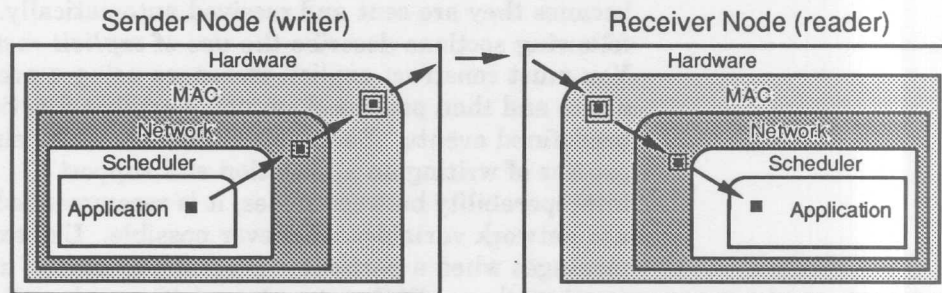


Figure 4-1. Sending a Message ■ = data

Implicit Messages: Network Variables

Figure 4-1 illustrates what happens when a node assigns a value to an output network variable. First, the application program assigns a value to the network variable. The scheduler then builds a network variable message and passes the message to the Network layer. The Network layer adds addressing information to the network variable message, then passes the message to the MAC layer. The MAC layer adds more information to the network variable message, and then sends the message over the communications channel.

When the network variable message is received by a node that reads that network variable, the message is unpackaged, as follows. First, the MAC layer validates the message. The Network layer then checks the addressing information contained in the message to see if it is intended for this node. If it is, it passes the network variable information to the scheduler. The scheduler then makes the new value available to the application program.

These messages are referred to as *implicit messages* because they are sent and received automatically. The following sections describe the use of *explicit messages*. You must construct explicit messages using a predefined object and then process them using explicit functions and predefined events. Because network variables simplify the process of writing an application and support interoperability between nodes, it is recommended that you use network variables whenever possible. Use explicit messages when a particular program, or part of a program, requires them. Both network variables and explicit messages can be combined in a single program.

Explicit Messages

The following sections describe how to use the objects, functions, and events used with explicit messages. The request/response mechanism, a special use of explicit messages, is described following the generalized description of explicit messages.

Following is a brief list of the steps described in the following sections. Objects, functions, and events are itemized for each section.

<i>Functional Step</i>	<i>NEURON C Feature</i>
1 Constructing a message	msg_out object
2 Sending a message	msg_send() function msg_cancel() function
3 Receiving a message	msg_arrives event msg_receive() function msg_in object
4 After sending a message with the ACKD service	msg_completes event msg_succeeds event msg_fails event
5 Sending a response to a message with the REQUEST service	resp_out object resp_send() function resp_cancel() function resp_arrives event resp_receive() function resp_in object
6 Allocating buffers explicitly	msg_alloc() function msg_alloc_priority() function msg_free() function resp_alloc() function resp_free() function

Constructing a Message

The name for the outgoing message object is `msg_out`. This definition is built into NEURON C. Use the `msg_send()` function to send the message. Only one outgoing message (or response) and one incoming message (or response) are available at any one time. For example, a program cannot build up two messages in parallel and send them both. Nor can two input messages be parsed at the same time.

The msg_out Object Definition

An outgoing message is predefined in the NEURON C Compiler as follows:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT, UNACKD, REQUEST}
    service_type;

struct {
    boolean priority_on;           // TRUE if a priority message
                                  // (default: FALSE)
    msg_tag tag;                  // message tag (required)
    int code;                     // message code (required)
    int data[MAXDATA];           // message data (default:none)
    boolean authenticated;        // TRUE if to be authenticated
                                  // (default: FALSE)
    service_type service;         // service type (default: ACKD)
    msg_out_addr dest_addr;       // see include file msg_addr.h
                                  // (optional)
} msg_out;
```

`priority_on`

is TRUE if the message is to receive priority when it is sent. Specify FALSE, or do not assign to this field, if the message is not a priority message. If used, this field must be the first field set in the message object. The default is FALSE (that is, nonpriority).

tag

is a user-specified identifier for the message. It is necessary to assign a value to the tag field every time a message is sent. Once the tag field has been assigned, the message must be either sent or cancelled. Message tags are used for two purposes: for addressing, and for correlating completion events and responses to particular messages.

For addressing, a message tag identifies the logical port of the node application from which a particular message is sent. The message tag is used to connect an outgoing message from a port on one node to a different port on another node. Message tags are required for all outgoing messages.

For correlation purposes, message tags are also used to qualify completion events and responses, for example:

```
when (msg_completes(tag1)).
```

By qualifying an event with a message tag, the event becomes TRUE only when an event corresponding to the outgoing message occurs.

A message tag declaration can optionally include connection information. The syntax for declaring a message tag is described in the *Declaring Message Tags* section, which follows this section.

code

is a numeric message code. This field is required. See the *Message Codes* section later in this chapter

data

Because of network buffer overhead, it is recommended that MAXDATA never exceed 228.

authenticated

NOTE: Do not use UNACKD or UNACKD_RPT in combination with authenticated messages. Use only the ACKD or REQUEST service type.

for a detailed description of numeric ranges for message codes.

is the data contained in the message. This field is optional; a message can consist of only a message tag and message code.

MAXDATA is a function of the pragma app_buf_out_size (see Chapter 6):

$\text{MAXDATA} = \text{app_buf_out_size} - 6$

or,

$\text{MAXDATA} = \text{app_buf_out_size} - 17$
(if explicit addressing is used)

Note that the NEURON CHIP firmware observes which locations in the data array have assignments and automatically sets the length of the outgoing message accordingly.

specify TRUE if the message is to be authenticated. Specify FALSE, or do not assign to this field, if the message does not need to be authenticated. The default is FALSE (that is, not authenticated).

specify one of the following:

ACKD (the default) - acknowledged service with retries

UNACKD - unacknowledged service

UNACKD_RPT- unacknowledged repeated service (message sent multiple times)

NOTE: To use this field, you must include the files `addrdefs.h` and `msg_addr.h`.

`dest_addr`

REQUEST- request/response service. When a message is sent using this service, the receiver node returns a response to the sender node, and the sender processes the response. The request/response mechanism is described in detail later in this chapter.

is an optional field in the `msg_out` object that the application program should assign a value to if the message is to be sent with explicit addressing. See the *Explicit Addressing* section later in this chapter for more information.

Declaring Message Tags

A message tag declaration can optionally include connection information. The syntax for declaring a message tag is:

msg_tag [*connection-info*] *tag-identifier* [, *tag-identifier* ...];

The *connection-info* field is an optional specification for connection options, in the form:

bind_info (*options*)

The connection options that apply to message tags are:

NOTE: It may not always be possible to determine `rate_est` and `max_rate_est`. For example, message output rates are often a function of I/O pins whose behavior is known only after a program has been run on a specific node. These values are used by a network management tool to perform network node analysis and are optional.

Although any value in the range 0-18780 may be specified, not all values are used. The values are mapped into encoded values `n` in the range 0-127. Only the encoded values are stored in the node's SI (self-identification) data. The actual value can be reconstructed from the encoded value. If the encoded value is zero, the actual value is undefined. If the encoded value is in the range 1-127, the actual value is $a = 2^{(n/8)} - 5$, rounded to the nearest tenth. (The actual value, a , produced by the formula, is in units of messages per second.)

`nonbind`

is a message tag which carries no addressing information and does not consume an address table entry. It may be used to permit a more optimal use of the NEURON CHIP resources when addressing is not required.

`rate_est (const-expr)`

is the estimated sustained message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780.

`max_rate_est (const-expr)`

is the estimated maximum message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780.

`tag-identifier`

is a NEURON C identifier for the message tag.

Message Codes

Message codes fall into the ranges shown in Table 4-1. Codes 0-62 and 64-78 are for use by applications. The lower range is typically used for application-specific messages, and the upper range is typically used by application gateways to other networks.

Table 4-1. Ranges for Message Codes

Application Messages	0 to 62	Generic application messages. The interpretation of the message code is left up to the application.
	63	Used only by responses. Indicates that the sender of the response was in an offline state and could not process the request.
Foreign Frames	64 to 78	Typically used by application gateways to other networks. The interpretation of the message code is left up to the application.
	79	Used only by responses. Indicates that the sender of the response was in an offline state and could not process the request.
Network Diagnostic Messages	80 to 95	See the <i>NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information</i> document (part no. 005-0018-01) for a description of these messages.
Network Management Messages	96 to 127	See the <i>NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information</i> document (part no. 005-0018-01) for a description of these messages.
Network Variables	128 to 255	The lower six bits of the message code contain the upper six bits of the network variable selector. The first data byte contains the lower eight bits of the selector. See the <i>NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information</i> document (part no. 005-0018-01) for a description of network variable encoding.

Example

An example of building a message is:

```
msg_tag motor;
#define MOTOR_ON 0
#define ON_FULL 100

msg_out.tag = motor;
msg_out.code = MOTOR_ON;
msg_out.data[0] = ON_FULL;
```

Block Transfers of Data

The NEURON C Compiler also handles a block transfer of message data into the `msg_out` or `resp_out` objects (see *Using the Request/Response Mechanism* section later in this chapter), through use of the `memcpy()` function. This is the only case where you can take the address of the `msg_out` or `resp_out` objects.

To copy a block of data into the `msg_out` object, the syntax is:

```
memcpy(msg_out.data, &s, sizeof(s));
```

The syntax is similar for the `resp_out` object.

<code>msg_out.data</code>	is the destination of the copy. This destination can also be a specific field of the message object (for example, <code>&msg_out.data[3]</code>).
<code>&s</code>	is a pointer to a structure containing the data to be copied. This field can be a pointer, an array, a pointer to an element of an array (<code>&a[5]</code>), and so on.
<code>sizeof(s)</code>	is the size of the source structure.

You can also use `memcpy()` to copy a block of data from the `msg_in` or `resp_in` objects. Note, however, that these are the only cases where you can take the address of the `msg_in` or `resp_in` objects.

`memcpy (&s, msg_in.data, sizeof (s));`

`&s` is a pointer to the destination structure. This field can be a pointer, an array, or a pointer to an element of an array (`&a[5]`).

`msg_in.data` is the source of the copy. This destination can also be a specific field of the message object (for example, `&msg_in.data[3]`).

`sizeof(s)` is the size of the destination structure.

An example of this block transfer of data follows:

```
msg_tag motor;

#define MOTOR_ON 0

typedef enum {
    MOTOR_FWD,
    MOTOR_REV
} motor_dir;

struct {
    long      motor_speed;
    motor_dir motor_direction;
    int       motor_ramp_up_rate;
} motor_on_message;

msg_out.tag = motor;
msg_out.code = MOTOR_ON;
motor_on_message.motor_direction = MOTOR_FWD;
motor_on_message.motor_speed = 500;
motor_on_message.motor_ramp_up_rate = 100;
memcpy (msg_out.data, &motor_on_message,
        sizeof(motor_on_message));
```

Sending a Message

The following functions control when messages are sent:

```
msg_send()
msg_cancel()
```

The `msg_send()` function has the following syntax:

```
void msg_send (void);
```

This function sends a message using the `msg_out` object.

The following code fragment illustrates sending a message:

```
msg_tag motor;
#define MOTOR_ON 0
#define ON_FULL 100           // (percent)

when (io_changes(switch1) to ON)
{
    // Send a message to the motor
    msg_out.tag = motor;
    msg_out.code = MOTOR_ON;
    msg_out.data[0] = ON_FULL;
    msg_send();
}
```

The `msg_cancel()` function has the following syntax:

```
void msg_cancel (void);
```

This function cancels the message being built for the `msg_out` object and frees the associated buffer, allowing another message to be constructed.

If a message is constructed but not sent before the task is exited, the message is automatically canceled.

Receiving a Message

A program usually receives a message through the predefined event when (`msg_arrives`). The `msg_receive()` function can also be used to receive a message.

msg_arrives Event

The predefined event for receiving a message is `msg_arrives`. Its syntax is:

`msg_arrives [(message_code)]`

If a message arrives, this event evaluates to `TRUE`. The event can optionally be qualified by a message code. In this case, the event is `TRUE` only when a message arrives containing the specified code.

When mixing unqualified `msg_arrives` events with qualified `msg_arrives` events, the `scheduler_reset` pragma must be specified so that the unqualified event when clause is processed after all the qualified event when clauses. A sample use of this event is shown in Listing 4-1.

```
#pragma scheduler_reset
when (msg_arrives (1))
{
    io_out(sprinkler, ON);
}

when (msg_arrives (2))
{
    io_out(sprinkler, OFF);
}

when (msg_arrives)           // default case for
                             // handling unexpected
                             // message codes
{
    // Do nothing, just discard it
}
```

Listing 4-1: Use of `msg_arrives` Event

msg_receive() Function

The `msg_receive()` function has the following syntax:

`boolean msg_receive(void);`

NOTE: When using the `msg_receive()` function, all messages are received in "raw" form, and thus, special events cannot be used. If special events were used, the application program would be required to check for these events explicitly via message code checks. For these reasons, it is recommended that `msg_receive()` not be used if the application program handles any special events (i.e. wink, online, and offline).

This function receives a message into the `msg_in` object. The function returns `TRUE` if a new message is received, otherwise it returns `FALSE`.

If no message is received, this function does not wait for one. A program may need to use this function if it wants to receive more than one message in a single task, as in bypass mode. If there already is a "received" message, it is discarded (that is, its buffer space is freed).

Calling `msg_receive()` or `resp_receive()` has the side-effect of calling `post_events()`. Thus, a call to `msg_receive()` or `resp_receive()` defines a critical section boundary (see the *Receiving a Response* section later in this chapter).

Format of an Incoming Message

NOTE: The fields of the `msg_in` object cannot be assigned to.

The name for the incoming message object is `msg_in`. This definition is built into NEURON C. A message is read by examining the appropriate fields in the object.

An incoming message is predefined in the NEURON C compiler as follows:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT, UNACKD, REQUEST}
    service_type;
```

```
struct {
    int code;                // message code
    int len;                 // length of message data
    int data[MAXDATA];       // message data
    boolean authenticated;   // TRUE if message was
                             // authenticated
    service_type service;    // service type used to send message
    msg_in_addr addr;        // see msg_addr.h include file
} msg_in;
```


code	is a numeric message code. See the <i>Message Codes</i> section earlier in this chapter for a detailed description of numeric ranges.
len	is the length of the message data.
data	is the data contained in the message. This field is valid only if len is greater than 0. MAXDATA is a function of the pragma app_buf_in_size (see Chapter 6): MAXDATA = app_buf_in_size - 6 or, MAXDATA = app_buf_in_size - 17 (if explicit addressing is used)
authenticated	is TRUE if the message was authenticated. This field is FALSE if the message was not authenticated.
service	is one of the following: ACKD- acknowledged service with retries UNACKD - unacknowledged service UNACKD_RPT- unacknowledged repeated service (message sent multiple times) REQUEST- request/response service. When a message is sent using this service, the receiver node returns a response to the sender node, and the sender processes the response. The request/response mechanism is described later in this chapter.

NOTE: Is is up to the application to reject messages which have failed authentication. Also note that the field name `authenticated` may be abbreviated as `auth`.

NOTE: To use this field, you must include the files `addrdefs.h` and `msg_addr.h`.

`addr` is an optional field in the incoming message that an application program may look at to determine the source and destination of the message. You can find the definition of the type `msg_in_addr` in the `msg_addr.h` include file.

Importance of a "Default" When Clause

Listing 4-1 (shown earlier in this chapter) illustrates an important technique to be used with messages: *any program that uses the explicit message facility to receive messages should be prepared to receive unwanted messages and discard them*. Discards can take the form shown in Listing 4-1, or could be a default case in a switch statement.

If a message were to arrive and the application fail to process it, that message would remain at the head of the queue forever, blocking the arrival of any other messages or network variable events and locking up the node forever until it is reset. One example of a message that would be sent to all nodes, most of which are not interested in the message, is the service pin message. Probably only a network management tool would want to process the service pin message; all other nodes would discard the message.

If a program does not process messages (either implicitly through the use of `when(msg_arrives)` or explicitly through the use of `msg_receive()`), the scheduler will automatically discard all incoming messages.

Note that a node that uses *only* network variables need not be concerned with this phenomenon, since the scheduler then handles *all* incoming messages, both implicit and explicit.

Example

The following example shows how the lamp/switch example in Chapter 1 could be rewritten to use explicit messages instead of network variables.

Lamp Program

First, here is the program for the lamp nodes:

```
// lamp.nc - Generic program for a lamp
// The lamp's state is governed by an incoming
// explicit message

#pragma scheduler_reset
#define LAMP_ON 1
#define LAMP_OFF 2
#define OFF 0
#define ON 1

// I/O declaration
IO_0 output bit io_lamp_control;

// Event-driven code
when (msg_arrives(LAMP_ON))
{
    // Turn on the lamp
    io_out(io_lamp_control, ON);
}

when (msg_arrives(LAMP_OFF))
{
    // Turn off the lamp
    io_out(io_lamp_control, OFF);
}

when (msg_arrives)
{
    // Just ignore any other message codes
}
```

Switch Program

Here is the program for the switch nodes:

```
// switch.nc - Generic program for a switch
// Send a message when the switch changes state

#define LAMP_ON 1
#define LAMP_OFF 2

// I/O Declaration
IO_4 input bit io_switch_in;

// Message tag declaration
msg_tag TAG_OUT;

// Event-driven code
when (reset)
{
    io_change_init(io_switch_in);
}

when (io_changes(io_switch_in))
{
    // Set up message code based on the switch state
    msg_out.code = (input_value == ON) ? LAMP_ON : LAMP_OFF;
    // Set up message tag and send message
    msg_out.tag = TAG_OUT;
    msg_send();
}
```

Connecting Message Tags

Every node has a default MSG_IN input message tag. Message tags may be connected to the the MSG_IN input message tag. For example, message tags on the two nodes are connected as follows:

TAG_OUT	connects to	MSG_IN
on the switch		on the lamp
node		node

Explicit Addressing

Explicit addressing of explicit messages and network variables may be implemented through the use of the include files `msg_addr.h` and `addrdefs.h`. (`addrdefs.h` must be included prior to `msg_addr.h`.) To use explicit addressing for outgoing messages, a program must assign appropriate values to all applicable fields of one of the elements of the `dest_addr` union in the `msg_out` object, prior to calling `msg_send()`. The message still needs a message tag, although no addressing information will be derived from the message tag. Thus, no matter how the message tag is bound, explicit addressing will override the tag.

When an explicit destination address is assigned by the application program, the message tag is only relevant for correlation with response and completion event processing. However, use of a standard message tag will still consume an address table entry, even if it is only used for messages whose addresses are explicitly set. To permit a more optimal use of NEURON CHIP resources, "non-bindable" message tags which carry no addressing information and do not consume an address table entry may be used. The declaration of a "non-bindable" message tag is accomplished with the following syntax:

```
msg_tag bind_info(nonbind[, <other info>]) <tag_name>;
```

See the *Declaring Message Tags* section earlier in this chapter for a more detailed discussion of the `nonbind` option. Also note that the use of explicit addressing has an effect on the buffer sizes needed by the NEURON CHIP. See Table 6-1, *Values for Buffer Sizes and Counts*, in Chapter 6 for more detailed information.

Network variable updates can be sent using explicit addressing by creating an explicit message that corresponds to a network variable update and explicitly setting the destination address. See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the explicit message format of a network variable update and Section A.3, *The*

Address Table, in the same document for more information on addressing.

Sending a Message with the ACKD Service

When a node sends a message using the *ACKD* service (the default), all receiver nodes must acknowledge receipt of the message to the sender node. As shown in Figure 4-2, the network processor is responsible for sending back the acknowledgment. This acknowledgment message contains no data and is sent to the network processor on the node where the message originated.

Note that the application layer plays no part in the acknowledgment of a message. How then does a program ever learn whether a message has succeeded or failed? The following section answers this question.

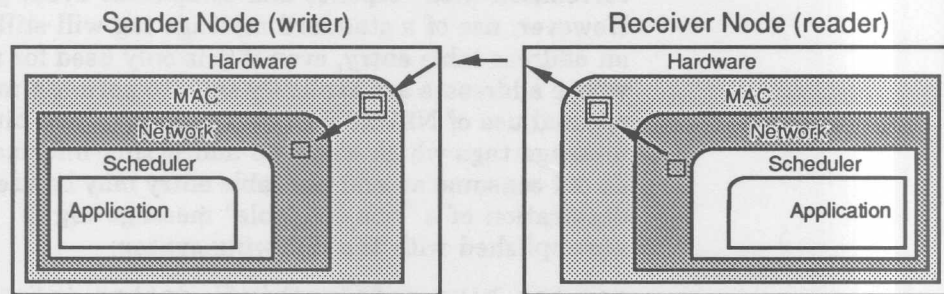


Figure 4-2. Acknowledging a Message

Message Completion Events

The following events can be used by the sender node to check the message completion status:

`msg_completes [(msg-tag-name)]`

`msg_succeeds [(msg-tag-name)]`

`msg_fails [(msg-tag-name)]`

All three events can be qualified by a message tag name. If unqualified, the event applies to any message.

msg_completes is the most general event. When an outgoing message completes (that is, succeeds or fails), this event evaluates to TRUE.

NOTE: See also the 'Comparison of resp_arrives and msg_succeeds' section later in this chapter.

The msg_succeeds event evaluates to TRUE when a message is successfully sent. The msg_fails event evaluates to TRUE when a message fails to be sent (after all retry attempts). (See Table 4-2 for a more precise breakdown of what "success" and "failure" mean for each service type.) For a given message, only *one* of these events evaluates to TRUE. The order of processing is thus important. If a message_completes event precedes the message_succeeds and message_fails events, the message_succeeds and message_fails events will never evaluate to TRUE.

These events are primarily of interest when a message is sent with either the ACKD service or the REQUEST service (see the *Using the Request/Response Mechanism* section later in this chapter). If a message is sent with the UNACKD or UNACKD_RPT service, the msg_succeeds and msg_completes events are *always* TRUE as soon as the message is transferred from the network processor to the Media Access Control (MAC) processor on the sender node.

Table 4-2. Success/Failure Completion Events

Service Used	SUCCESS =	FAIL=
UNACKD	Message is transmitted to MAC processor.	*
UNACKD_RPT	<i>N</i> messages are transmitted to MAC processor. (<i>N</i> is the number of repeats.)	*
ACKD	All acknowledgments have been received by the Network processor on the sender node.	One or more acknowledgments are not received. This applies to both messages and network variables.
REQUEST	All responses have been received by the Application processor on the sender node.	<p><i>For messages:</i> One or more of the responses did not arrive.</p> <p><i>For network variable poll:</i> (a) One or more of the responses did not arrive. (b) None of the responses had valid data.</p>
<p><i>*In all cases, if the NEURON CHIP firmware encounters an addressing error, a failure event occurs (see Appendix F). If a network variable or message is unbound, a success event occurs.</i></p>		

Processing Completion Events for Messages

When you send a message, it is optional whether you check the completion event. Several restrictions apply, however, if you do check the completion event.

First, if you check for either `msg_succeeds` or `msg_fails`, you must check for *both* events. The alternative is simply to check for `msg_completes`.

Second, if you qualify a completion event with a particular message tag, then you must *always* process completion events for that message tag. A program can thus process completion events for some of its message tags, and ignore completion events for other message tags. In the following example, completion events for TAG1 are processed, and completion events for TAG2 are *not* processed:

```
when (io_changes(dev1))
{
    .
    .
    .
    msg_out.tag = TAG1;
    .
    .
    .
    msg_send();
}

when (msg_completes(TAG1))
{
    .
    .
    .
}

when (io_changes(dev2))
{
    .
    .
    .
    msg_out.tag = TAG2;
    .
    .
    .
    msg_send();
}
```

A third restriction applies to use of the *unqualified* completion event, which implicitly refers to *all* messages. When the unqualified completion event is used, all acknowledged messages must be processed, either explicitly for each message tag, or implicitly through use of an unqualified event each time a message is sent.

The following code shows correct processing of completion events by message tag:

```
int failures[2], success;
msg_tag TAG1, TAG2;

when (io_changes(toggle))
{
    msg_out.tag = TAG1;
    msg_out.code = TOGGLE_STATE;
    msg_out.data[0] = input_value;
    msg_send();

    msg_out.tag = TAG2;
    msg_out.code = TOGGLE_STATE;
    msg_out.data[0] = input_value;
    msg_send();
}

when (msg_fails(TAG1))
{
    failures[0]++;
}

when (msg_fails(TAG2))
{
    failures[1]++;
}

when (msg_succeeds)      // any message qualifies
{
    success++;
}
```

Preemption Mode and Messages

The system enters *preemption mode* when there is no buffer available for an outgoing message. If the system needs a free buffer, it causes the application program to wait and processes only completion events, responses, and incoming network variables and messages to facilitate buffers becoming free. No other predefined or user-defined events are processed. The watchdog timer is automatically updated during this wait. If the program waits for more than a configurable number of seconds, the node is reset. (This

should happen only under extreme network congestion or network failure.)

With network variables, the system only enters preemption mode if:

- synchronous network variables are used, or
- `flush_wait()` is called.

Once the system is in preemption mode, further attempts to send a message from a task associated with a `when` clause using a message completion event `when` clause will cause a node reset if no buffer is available for the new message. The following sequence is therefore *not* recommended:

```
when (TOGGLE_ON)
{
    // build a message
    // send the message
}

when (msg_completes)
{
    msg_out.tag = t;           // This sequence is not
                              // recommended.
    msg_out.code = 1;         // Causes a node reset
                              // if the system is
                              // already in preemption
                              // mode
}
```

Instead of using this sequence, messages should be built and `msg_send()` called in a task with a `when` clause that does not use the `msg_completes` event. When synchronous network variables (or messages with implicit buffer allocation) are used, there is a good chance that preemption mode will be entered when network congestion occurs. When preemption mode is entered, **no** outgoing network variable updates occur, priority or otherwise. Thus, a program that expects priority updates to occur within a bounded amount of time should **not** use nonpriority synchronous network variables or messages with implicit buffer allocation.

To allocate and free buffers explicitly, use the functions described later in this chapter in the *Allocating Buffers Explicitly* section.

Asynchronous and Direct Event Processing

Events can be checked using when clauses and events such as when (msg_completes), when (msg_fails), and when (msg_succeeds). This type of event processing is referred to as *asynchronous processing*, since the scheduler handles the exact order of execution. An alternate technique is *direct event processing*, in which completion events are checked inside tasks, with if and while statements.

The following example indicates one way asynchronous and direct processing *cannot* be combined. Do not include message completion events in a task associated with a message completion event clause:

```
when (msg_completes)
{
    post_events();
    if (msg_completes)    // not recommended
        x = 4;
}
```

Asynchronous event processing can be used in programs that also do direct event processing. Asynchronous event processing is the typical method for processing events. This method results in smaller application programs. The application program should call the flush_wait() function before the transition from asynchronous to direct event processing. The flush_wait() function ensures that all outstanding completion events and response events are processed before switching to direct event processing.

Here is an example of sending a message and processing the completion event directly (that is, checking the event inside a task rather than inside a when clause):

```
msg_tag motor;
#define MOTOR_ON 0

when (x==3)
{
    // send a message
    flush_wait();
    msg_out.tag = motor;
    msg_out.code = MOTOR_ON;
    msg_send();

    // check completion status
    while (!msg_succeeds(motor)) {
        post_events();
        if (msg_fails(motor))
            node_reset();
    }
}
```

If you need to switch from asynchronous processing to direct processing within a program, use the `flush_wait()` function to ensure that all outstanding completion events and response events are processed before the program begins processing messages and network variables directly.

Using the Request/Response Mechanism

Request/response messages provide a mechanism for an application running on one node to request data from an application running on another node. The request/response mechanism is used automatically by the firmware to poll input network variables and can also be used by application programs that use explicit messaging.

A *request* is a message that uses the *request* service. Sending a request message is similar to polling a network variable. A poll receives the most recent value from the scheduler for a particular network variable. A request, in contrast, can force the application on the responding node to

evaluate the variable *at the time of the request* and then send back a response.

The functions, events, and objects for constructing, sending, and receiving responses are analogous to those for constructing, sending, and receiving messages, described in the previous section. They are also summarized in the following paragraphs.

An example of sending a request is:

```
msg_tag motor;
#define MOTOR_STATE 1

when (io_changes(switch1) to 2)
{
    //send a request to the motor
    msg_out.tag = motor;
    msg_out.service = REQUEST;
    msg_out.code = MOTOR_STATE;
    msg_send();
}
```

The request is packaged as shown in Figure 4-1. The application program on the receiver node receives the request through a when clause (or `msg_receive()` function) and must then formulate a response to this request, as shown in Figure 4-3.

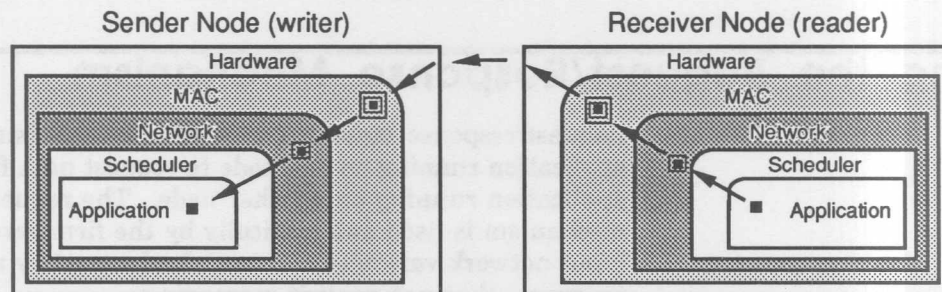


Figure 4-3. Sending a Response ■ = data

Constructing a Response

Note in Figure 4-3 that the response contains a data portion that is sent to the application processor of the sender node. A response is different from an acknowledgment (Figure 4-2), which does not contain a data portion and is sent only to the network processor on the sender node.

The name of the outgoing response object is `resp_out`. The response inherits its priority and authentication designation from the request to which it is replying. Because the response is returned to the origin of the request, no message tag is necessary. For the same reason, there is no explicit addressing feature for a response.

The outgoing response object is predefined in the NEURON C Compiler as follows:

```
struct {  
    int code;           // message code  
    int data[MAXDATA]; // message data  
} resp_out;
```

code is a numeric message code. This field is required. See the *Message Codes* section earlier in this chapter for a detailed description of numeric ranges.

data is the data contained in the message. This field is optional. `MAXDATA` is a function of the pragma `app_buf_in_size` (see Chapter 6):

`MAXDATA = app_buf_in_size - 6`

or,

`MAXDATA = app_buf_in_size - 17`
(if explicit addressing is used)

Note that the system observes which locations in the data array have

assignments and automatically sets the length of the outgoing response accordingly.

Sending a Response

A response is sent with the `resp_send()` function. Responses must be sent from the same critical section that processed the incoming request. Once response construction has started, the incoming request can no longer be examined. Also, no other intervening messages can be sent or received.

The syntax for `resp_send()` is:

```
void resp_send (void);
```

This function sends a response using the `resp_out` object.

Receiving a Response

A program usually receives a response through the predefined event when (`resp_arrives`). The `resp_receive()` function can also be used to receive a response.

resp_arrives Event

The predefined event for receiving a response is `resp_arrives`.

Its syntax is:

```
resp_arrives [(msg-tag-name)]
```

If a response arrives, this event evaluates to `TRUE`. The event can optionally be qualified by a message tag name.

resp_receive() Function

The resp_receive() function has the following syntax:

```
boolean resp_receive(void);
```

This function receives a response into the resp_in object. The function returns TRUE if a response is received, otherwise it returns FALSE. The response is automatically discarded at the end of the task that receives it.

Calling resp_receive() has the side-effect of calling post_events(), so a call to resp_receive() defines a critical section boundary.

Format of a Response

The name of the incoming response object is resp_in.

The incoming response structure is predefined in the NEURON C Compiler as follows:

```
struct {  
    int code;           // message code  
    int len;           // length of message  
                        // data  
    int data[MAXDATA]; // message data  
    resp_in_addr addr; // explicit address  
                        // see the include file  
                        // msgaddr.h  
} resp_in;
```

code is a numeric message code. See the *Message Codes* section earlier in this chapter for a detailed description of numeric ranges.

len is the length of the message data.

data

is the data contained in the message. This field is valid only if len is greater than 0. MAXDATA is a function of the pragma app_buf_in_size (see Chapter 6):

MAXDATA = app_buf_in_size - 6

or,

MAXDATA = app_buf_in_size - 17
(if explicit addressing is used)

NOTE: To use this field, you must include the files `addrdefs.h` and `msg_addr.h`.

addr

is an optional field in the incoming message that an application program may look at to determine the source and destination of the message. You can find the definition of the type `resp_in_addr` in the `msg_addr.h` include file.

Examples

This example shows sending a request and asynchronously receiving the responses. (The code for receiving this request and responding to it follows.)

```
when (io_changes(toggle))
{
    msg_out.tag = TAG1;
    msg_out.code = DATA_REQUEST;
    msg_out.service = REQUEST;
    msg_send();
}

when (resp_arrives(TAG1))
{
    if (resp_in.code == OK)
        process_response(resp_in.data[0]);
}
```


Here is the code for the responder to this request:

```
when (msg_arrives(DATA_REQUEST))
{
    int x, y;
    x = msg_in.data[0];
    y = get_response(x);
    resp_out.code = OK; // msg_in no longer
                        // available
    resp_out.data[0] = y;
    resp_send();
}
```

The following example shows sending a request and receiving the responses directly:

```
int x;
msg_tag motor;
#define MOTOR_ON 0
#define DO_MOTOR_ON 3

when (command == DO_MOTOR_ON)
{
    // send a request
    msg_out.tag = motor; // construct the
                        // message

    msg_out.code = MOTOR_ON;
    msg_out.service = REQUEST;
    msg_send(); // send the
                // message

    // wait for completion
    while (!msg_succeeds (motor)) {
        post_events();
        if (msg_fails(motor))
            node_reset();
        else if (resp_arrives(motor)) {
            x = x + resp_in.data[0];
            resp_free(); // optional
        }
    }
}
```

Comparison of `resp_arrives` and `msg_succeeds`

Use both `resp_arrives` and the completion events (`msg_succeeds`, `msg_fails`, `msg_completes`) for the same transaction because these events give you different information. The following example illustrates this difference.

Suppose you send one request to six nodes. Three of the responses are received and three are not received. In this case, the `resp_arrives` event will be `TRUE` three times, once each time a response arrives. The `msg_succeeds` event will never become `TRUE`, because some of the responses did not arrive. The `msg_fails` event will become `TRUE` when the allotted time for all responses to arrive is exceeded. (In other words, for `msg_succeeds` to be `TRUE`, all of the responses must be received.)

Responses always arrive before the message completion codes (`msg_completes`, `msg_fails`, `msg_succeeds`).

Buffers

NOTE: If your program is linked for a NEURON 3120 CHIP, the linker will adjust the output buffer defaults to one priority and one nonpriority buffer. The number of input buffers will still default to 2.

The number of incoming and outgoing buffers available for use by a NEURON CHIP program can be set at compile time. The default is two priority output buffers, two nonpriority output buffers and two input buffers. See Chapter 6 for a discussion of buffer allocation. For most efficient response, set the number of buffers to equal the expected number of responses. If a disproportionately large number of responses (for example, 10) are expected for the same request, some responses may never be received if only a limited number of buffers are available.

Note also that the same pool of buffers is used for processing both incoming messages and responses. If you are processing events directly (that is, bypassing the services of the scheduler), be sure to check for messages as well as responses so that messages are processed and buffers are freed up regularly.

Allocating Buffers

Normally, when you build a message, a buffer is allocated automatically, and when you leave a critical section, any outstanding message buffer is freed by the scheduler automatically.

The following functions allow you to allocate and free message buffers explicitly:

boolean msg_alloc (void);

boolean msg_alloc_priority (void);

void msg_free (void);

A message travels along one of two paths out of the NEURON CHIP: the priority path or the nonpriority path. As its name suggests, the priority path has precedence over the nonpriority path. Thus, if you allocate a buffer out of the priority buffer pool, the message is more likely to succeed on a congested network.

The `msg_alloc()` and `msg_alloc_priority()` functions return **TRUE** if a `msg_out` object can be allocated.

Otherwise, these functions return **FALSE**. A program needs to call one of these functions if it does not want to wait for a buffer. If the function returns **FALSE**, the program can choose to go off and do something else, then try again later.

The `msg_alloc_priority()` function allocates a priority buffer. The `msg_alloc()` function allocates a nonpriority buffer. If you are using the system default, up to two buffers of each type can be in use at the same time. (Note that you can allocate up to the maximum number of buffers set by the `pragma`.)

The `msg_free()` function frees the `msg_in` object. You do not normally need to free a message buffer, since this is done for you when you exit a task. However, you might want to free a buffer explicitly if you are finished with it in a task, but you have more work to do before exiting the task.

Normally an output buffer is allocated by assigning a value to one of the fields of the `msg_out` object. In the event that a buffer is not available, application processing will be suspended (preemption mode) until one is available. If you want to avoid possibly suspending processing, use `msg_alloc()`. If no buffer is available, a `FALSE` value will be returned, and processing continues. This allows the application to do something else in the event that there are no outgoing buffers available, rather than stopping to wait for a buffer.

An input buffer is normally freed at the end of the critical section in which the `msg_receive()` call is made. The application may choose to free the buffer earlier than this by calling `msg_free()`. After this call, the `msg_in` object no longer contains the received message, but the network processor is able to use the freed buffer for another incoming message. Note that `msg_alloc()` and `msg_free()` are unlike standard memory allocation functions. A buffer allocated by `msg_alloc()` is not freed by a matching call to `msg_free()`. Instead, a `msg_send()` automatically frees a buffer allocated by `msg_alloc()`, and a `msg_free()` automatically frees a buffer allocated by `msg_receive()`.

The analogous functions for allocating and freeing responses are:

```
boolean resp_alloc (void);  
void resp_free (void);
```

The following example shows a task that creates two messages. Before creating and sending each message, the code checks buffer availability with `msg_alloc()`.

```
msg_tag motor1;
msg_tag motor2;
#define MOTOR_ON 0

when (x == 2)
{
    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor1;
    msg_out.code = MOTOR_ON;
    msg_send();

    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor2;
    msg_out.code = MOTOR_ON;
    msg_send();
}
```


5

Additional Features

This chapter describes additional features in NEURON C. It describes the scheduler reset mechanism in more detail. In special cases requiring a scheduling algorithm different from that of the NEURON CHIP firmware scheduler, you may want a program to run in bypass mode and use the `post_events()` function, also described here. Other topics discussed in this chapter include sleep mode, error handling, and status reporting.

The Scheduler

Chapter 2 introduced the basic functioning of the NEURON CHIP firmware scheduler, shown in Figure 5-1. You will recall that priority when clauses are executed in the order specified every time the scheduler runs. If any priority when clause evaluates to TRUE, its task is run and the scheduler starts over. If none of the priority when clauses evaluates to TRUE, then a nonpriority when clause is evaluated, selected in a round-robin fashion. If the when clause is TRUE, its task is executed. If the nonpriority when clause is FALSE, its task is ignored. In either case, the scheduler returns to the top of the loop.

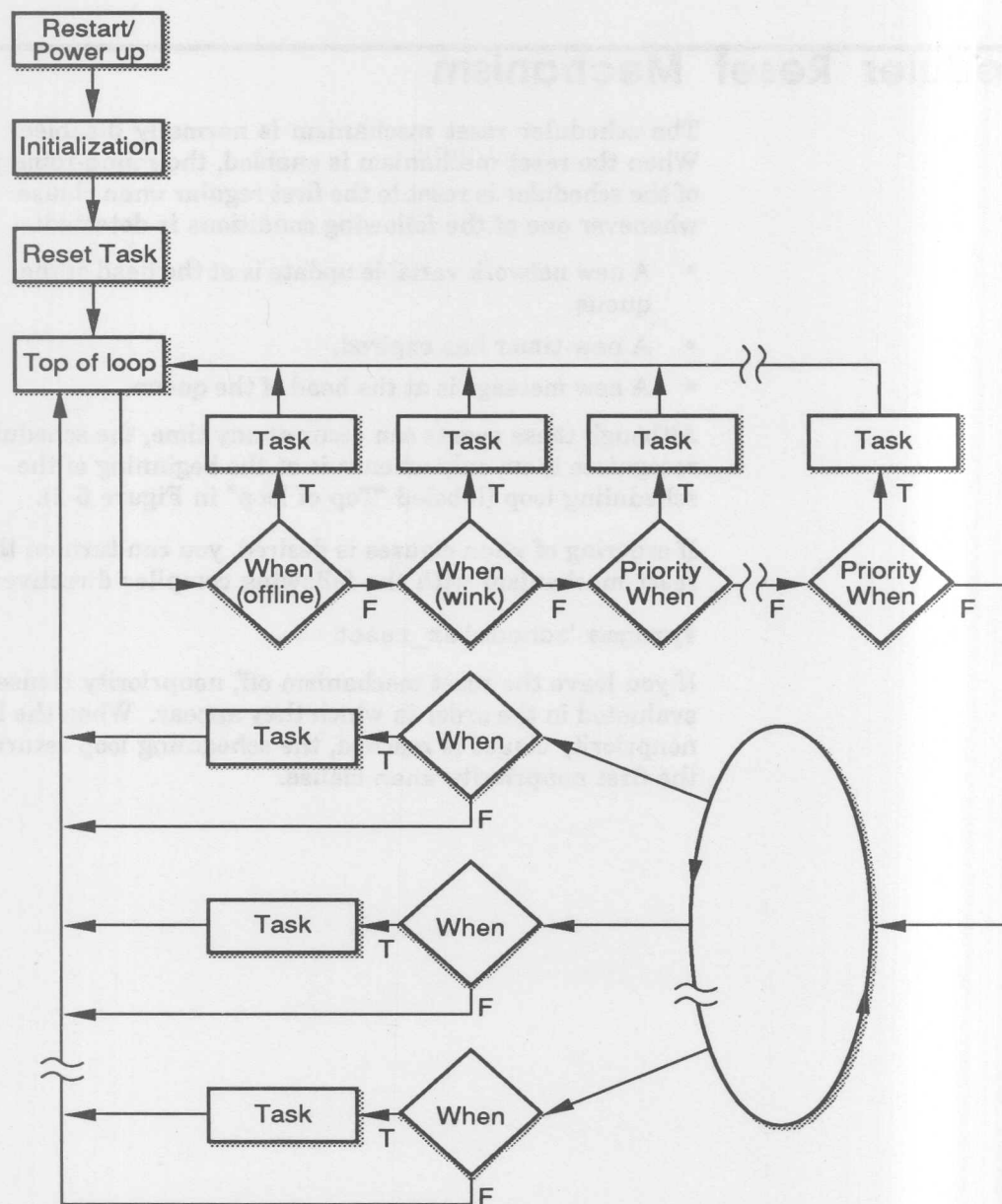


Figure 5-1. NEURON CHIP Firmware Scheduling of Nonpriority and Priority When Clauses

Scheduler Reset Mechanism

The scheduler reset mechanism is normally disabled. When the reset mechanism is enabled, the round-robin part of the scheduler is reset to the first regular when clause whenever one of the following conditions is detected:

- A new network variable update is at the head of the queue.
- A new timer has expired.
- A new message is at the head of the queue.

Although these events can occur at any time, the scheduler recognizes them only when it is at the beginning of the scheduling loop (labeled “Top of loop” in Figure 5-1).

If ordering of when clauses is desired, you can turn on the reset mechanism with the following compiler directive:

```
#pragma scheduler_reset
```

If you leave the reset mechanism off, nonpriority clauses are evaluated in the order in which they appear. When the last nonpriority clause is reached, the scheduling loop returns to the first nonpriority when clause.

Example

Turning on the reset mechanism ensures that events are processed in the order intended. For example, you may want to be sure that specific events are checked for first, followed by a “catch-all” event, as illustrated in this fragment:

```
#pragma scheduler_reset
network input int NV1, NV2, NV3, NV4;

when (nv_update_occurs(NV1))
{
    .
    .
    .
}
when (nv_update_occurs(NV2))
{
    .
    .
    .
}
when (nv_update_occurs)
{
    // provides a generic check
    // for all network variable
    // updates
    .
}
```

Note that updates received for NV1 would cause both the first and third events to become TRUE. Similarly, updates for NV2 would cause the second and third events to become TRUE. It is thus important that these when clauses be evaluated in their given order after a network variable update. Using `scheduler_reset`, the `nv_update_occurs` event for NV1 is always checked first whenever a new network variable update is at the head of the queue.

Bypass Mode

All scheduling of NEURON C programs, as described above, is event-driven and handled by the scheduler. Within a program, however, you can choose when to return control to the scheduler. The term *bypass mode* refers to a method of programming in which one when clause always evaluates to TRUE and never returns. In this case, a single task must handle all event processing.

You will use bypass mode rarely, and only in cases where you need a different scheduling algorithm than that provided by the NEURON CHIP firmware scheduler. While in bypass mode, the program is responsible for all event processing. You define critical sections through the `post_events()` function (see the following section), and then check for predefined events in `if`, `while`, and `for` expressions.

post_events() Function

Use `post_events()` to define a boundary of a critical section at which network variable updates are sent and incoming network variable updates are processed.

NOTE: This function is automatically called at the top of the scheduling loop.

When the `post_events()` function is called, a number of things happen:

- Any outgoing network variable updates are sent. In the case of synchronous network variables, all the updates are sent. For nonsynchronous network variables, as many updates are sent as buffers are available. Any unsent updates will be sent the next time `post_events()` is called.
- Incoming network variable update events are received.
- New messages are examined.
- Timers are examined to see if they have expired.
- The watchdog timer is reset (to keep it from timing out). See the following section on the watchdog timer.

The `post_events()` function can be used to improve network performance by calling it immediately after modifying network variables. Calling `post_events()` signals the network processor to start execution of the formatting of the outgoing packet before the when task completes, thus utilizing the NEURON CHIP multi-processor architecture to its fullest.

Watchdog Timer

The watchdog timer times out within the range of .84 to 1.68 seconds with a 10 MHz input clock. (This value scales with the input clock.) Normally, the scheduler ensures that the watchdog timer is reset periodically. If a program enters a very long task, however, the watchdog timer may expire, which causes a node reset.

To ensure that the watchdog timer does not expire, call the `watchdog_update()` function periodically within long tasks (or when in bypass mode). The `post_events()`, `msg_receive()`, and `resp_receive()` functions also update the watchdog timer, as well as the pulsecount output object.

An example of using the `watchdog_update()` function is:

```
when (TRUE)
{
    post_events();
    if (nv_update_occurs (NV1)){
        .
        .
        .
    } else if (nv_update_occurs (NV3)){
        . // long task
        .
        .
        watchdog_update();
        . // more long task
        .
        .
    }
}
```

Additional Predefined Events

The following three predefined special events result from network management messages:

offline

online

wink

The offline event occurs when an offline network management command is received from a network management tool. This event is always handled as the first priority when clause. The online event occurs when an online message is received from a network management tool. The wink event occurs when a wink command is received from a network management tool.

The offline event can be used to place a node offline in case of an emergency, for maintenance, or in response to some other system-wide condition. Once offline, a node will respond only to network management commands until reset or brought back online. (Reset can occur by physically resetting the node by grounding the NEURON CHIP reset line, or through a reset network management command.) After execution of the task associated with an offline when clause, the application program does not run until the node is either reset or brought back online.

A simple use of these two events would be:

```
when (offline)
{
    x(); // Clean up before going offline.
} // Node goes offline here; application
    // program stops running.

when (online)
{
    y(); // Start up again (poll inputs,
        // and so on)
}
```

Going Offline in Bypass Mode

Use the `offline_confirm()` function if the offline event is checked for outside of a `when` clause, as in bypass mode. The `offline_confirm()` function sets the state of the node to offline and returns immediately. Use of this function allows a node to confirm that it has finished its cleanup and is now going offline.

As shown below, in bypass mode, the program continues to run even though the node is offline. In bypass mode, it is up to you to determine which events are not processed when the node is offline.

Here is an example of using `offline_confirm()` in bypass mode:

```
when (TRUE)
{
    while (TRUE) {
        post_events();
        if (online)
            break;
        if (nv_update_occurs) {
            .
            .
            .
        } else if (offline) {
            x();
            offline_confirm();
            // Wait for online
            while (!online) {
                post_events();
            }
        } else {
            ...
        }
    }
}
```

Wink Event

The wink event allows a node to perform an action in response to a network management wink command. The wink event becomes TRUE any time a wink command is received by a node, whether configured or unconfigured.

In an unconfigured node, I/O and variable initialization occur before the wink event is evaluated. However, none of the initialization in the when (reset) task has occurred. In addition, the scheduler is not running on an unconfigured node, so events can be processed only through direct event processing. Updates to network variables and messages are not sent because the node is unconfigured. Timer objects cannot be used within the wink task.

See also the delay() function description in Chapter 2.

Sleep Mode

Sleep mode places the NEURON CHIP in a low-power state. A program can instruct the NEURON CHIP to enter sleep mode at any time through use of a two-step process. The first step involves flushing the NEURON CHIP of all pending network variable updates as well as all outstanding outgoing and incoming messages. The second step actually puts the NEURON CHIP into sleep mode when the flush completes. The NEURON CHIP always wakes up when the service pin is pressed, or when there is activity on an I/O pin (the pin selected is configurable) or on the communications channel, or both.

The following code fragment illustrates this process:

```
mtimer m_30;
network output int NV1;
static int temp;

when (timer_exp(m_30))
{
    NV1 = temp;
    flush(TRUE);
}
when (flush_completes)
{
    sleep(COMM_IGNORE);
}
```

Flushing the NEURON CHIP

The `flush()` function is used to instruct the NEURON CHIP to finish processing all outgoing and incoming messages. When the flush is complete, the `flush_completes` event becomes TRUE.

flush() and flush_cancel() Functions

The `flush()` function causes the NEURON CHIP to monitor the status of all outgoing and incoming messages. Its syntax is:

flush (boolean *comm_ignore*);

comm_ignore

specify TRUE if the NEURON CHIP is to ignore communications channel activity while it is flushing. Specify FALSE if the NEURON CHIP is to accept incoming messages. (This parameter should be the same as the `comm_ignore` parameter used with the `sleep()` function that follows the `flush`.)

While the flush is occurring, the program continues to run. The origination of new messages by the program while the flush is in progress simply delays the flush completion.

If the `comm_ignore` option is set to `TRUE`, new packets that arrive during the flush are discarded unless they are acknowledgments, responses, challenges, or replies.

The flush can be canceled by calling the `flush_cancel()` function.

flush_completes Event

The following predefined event becomes `TRUE` when the flush completes:

`flush_completes`

This event becomes `TRUE` when all outgoing network buffers and application buffers are free, no more incoming messages are outstanding, and no network variable updates are outstanding.

Putting the NEURON CHIP to Sleep

When the `flush_completes` event becomes `TRUE`, the NEURON CHIP can be put to sleep through use of the `sleep()` function. Its syntax is:

sleep (*flags* [, *io_object_name*])

flags

is one or more of the following three flags, or 0 if no flag is specified. If two or more flags are used, they are or'd together. The following flags can be specified:

COMM_IGNORE

causes incoming messages to be ignored.

PULLUPS_ON

enables all internal pullup resistors (the default action is to disable the pullups - this lowers power consumption).

TIMERS_OFF

turns off all timer objects (declared with `mtimer` and `stimer`) in the program.

io_object_name

is an input object declared for any one of pins IO_4 through IO_7. When any I/O transition occurs on the specified pin, the NEURON CHIP wakes up. If this parameter is omitted, I/O is ignored once the NEURON CHIP is in sleep mode. This I/O object can be defined for wakeup purposes only, or could be used for other I/O purposes as well.

For example, to sleep and turn off timers and enable pullups, the syntax is:

```
sleep(TIMERS_OFF | PULLUPS_ON)
```

The following example shows use of I/O pin 4 for wakeup:

```
IO_4 input bit wakeup_pin1;

when timer_expires(timer_2)
{
    sleep(COMM_IGNORE, wakeup_pin1);
}
```

A node can also be forced to sleep even though the flush has not completed, as described in the following section on *Forced Sleep*.

When an event occurs that wakes the NEURON CHIP, the program resumes at the first statement after the `sleep` function. If the `sleep()` call is the last statement in the task, the program returns to the scheduler.

A node will wake up whenever a packet is received by the transceiver (unless `COMM_IGNORE` is specified). This is true even if the packet is not addressed to the node. The application program is responsible for putting the node back to sleep when this occurs.

If a NEURON CHIP sleeps for less than the receive timer duration and uses the `COMM_IGNORE` option, it may receive duplicate messages or network variable update events. The default receive timer is set by a network management tool during or following node installation. During development, the receive timer duration for a node is set in the Node Create screen of the LONBUILDER network manager. See the *LONBUILDER User's Guide* for more information.

Forced Sleep

You can also force a node to sleep even though the flush is not complete. Under certain network conditions, such as extreme congestion, the flush could take a long time to complete. To avoid consuming too much power, the application can stop waiting for the flush to complete and sleep anyway.

To force a node to sleep, call the `sleep()` function without waiting for the `flush_completes` event. An example of forcing a node to sleep is:

```
...
    flush(TRUE);                // start flush; ignore
                                // incoming packets
    flush_timeout = 300;        // start flush timeout
                                // timer (300 msec)
}

when (timer_expires(flush_timeout))
when (flush_completes)
{
                                // sleep since either
                                // flush completed or
                                // timed out
    flush_timeout = 0;          // Turn off timer if not
                                // expired
    sleep(COMM_IGNORE);
}
```

When sleep is forced, the following occurs:

- 1 All pending network variable updates, outstanding application buffers, and outstanding network buffers are cleaned up.
- 2 If the `COMM_IGNORE` option is specified, any incoming network buffers are freed.
- 3 If any outstanding incoming application buffers remain, the node will fail to sleep (regardless of whether the `COMM_IGNORE` option was specified). This feature prevents the node from receiving stale messages when it wakes up. In the example above, the application would have 300 milliseconds to process any incoming messages already in the queue. In addition, since the `COMM_IGNORE` parameter was set to `TRUE` in the call to `flush()`, no new incoming messages would arrive. Thus, it is likely that the node will sleep, assuming it processes, in the 300 msec prior to the timeout, any incoming messages that were outstanding prior to the call to `flush()`.

Error Handling

An application has two primary options for recovering from errors: resetting the node and restarting the application. A third option is for the application to log errors without resetting at all.

Resetting the Node

A node can be reset by calling the `node_reset()` function. When this function is called, all processors (application, network, and MAC) on the NEURON CHIP are reset immediately. The reset pin of the NEURON CHIP is driven low, and this can be used to reset external transceivers and logic.

When a NEURON CHIP is reset, it executes the entire initialization sequence. The amount of time required for initialization is a function of the amount of off-chip memory as well as the size of the application program.

There are a number of disadvantages to resetting a node. First, when a node is reset, all state information not kept in EEPROM is lost. The network processor may receive duplicate packets. In addition, any packets that have been acknowledged by the network processor but not processed by the application are lost.

Restarting the Application

The `application_restart()` function is used to reset only the application processor. When this function is called, the firmware clears all timer objects and initializes I/O objects, network variables, and static variables and then executes the reset clause. Some synchronization cleanup is done before restarting the application. Any outgoing messages in progress are terminated. Incoming messages are unaffected. Outstanding completion codes and responses are discarded. An application restart does not lose network state information. Since only the application processor is reset, the network and MAC processors continue to process network traffic.

Logging Application Errors

The third option is to log application errors without resetting by calling the `error_log()` function, which is passed an error number between 1 and 127. This function writes the last number into a dedicated location in EEPROM. Network management tools can use the query status network diagnostic command to read the last error.

The NEURON C Debugger maintains a log of the last 25 error messages. On a NEURON Emulator, the NEURON CHIP firmware adds a delay of up to 70 msec between writes to the error log to give the PC time to retrieve the last value.

System Errors

System errors include programming errors and network errors and inconsistencies. System error numbers are logged using the same mechanism as application errors.

See Appendix F for an annotated list of system error messages.

Access to Node Error Status

An application has local access to the same diagnostic status information that is available to a network management tool. The status information is stored in the status structure, which is obtained through the `retrieve_status()` function. Its syntax is:

```
void retrieve_status (status_struct *status_p);
```

The fields of the status structure are described in detail in Appendix C in *Section C.2*. Use the `clear_status()` function to clear certain status structure fields (the statistics information, the reset cause register, and the error log).

Example

An example of using the `retrieve_status()` function follows.

```
#define unconfigured          0x02
#define config_on_line        0x04
#define config_off_line       0x0C
#define power_up_reset        0b1
#define power_up_reset_mask   0b1
#define external_reset         0b10
#define external_reset_mask    0b11
#define WDT_reset              0b1100
#define WDT_reset_mask         0b1111
#define SI_reset               0b10100
#define SI_reset_mask          0b11111

#include <status.h>

status_struct status; // structure type defined
                      // in status.h

unsigned long    transmission_errors;
unsigned long    transaction_timeouts;
unsigned long    receive_transaction_full;
unsigned long    lost_messages;
unsigned long    missed_messages;
unsigned long    reset_cause;
unsigned short   node_state;
unsigned short   version;
unsigned short   error_log;
unsigned short   model_number;

retrieve_status(&status);
    // obtain node status structure

transmission_errors = status.status_xmit_errors;
    // number of received packets with CRC errors
```



```

transaction_timeouts =
    status.status_transaction_timeouts;
// number of timeouts using Ackd or Req/Resp
// transactions

receive_transaction_full =
    status.status_rcv_transaction_full;
// number of times incoming message (other than
// Unackd) was lost due to receive transaction
// database overflow

lost_messages = status.status_lost_msgs;
// number of times incoming message was lost
// because there was no application buffer

missed_messages = status.status_missed_msgs;
// number of times incoming message was lost
// because there was no network buffer

reset_cause = status.status_reset_cause;

if ((reset_cause & power_up_reset_mask) ==
    power_up_reset) {
    // last reset was a power_up
}

if ((reset_cause & external_reset_mask) ==
    external_reset) {
    // last reset was from the NEURON RESET pin
}

if ((reset_cause & WDT_reset_mask) ==
    WDT_reset ) {
    // last reset was from the watchdog timer
    // timing out
}

if ((reset_cause & SI_reset_mask) == SI_reset ) {
    // last reset was software initiated by a
    // call to node_reset()
}

```

```

node_state = status.status_node_state;
if (node_state == unconfigured) {
    // this node has not been configured
}
if (node_state == configured_online) {
    // this node is running its application
}
if (node_state == configured_offline) {
    // this node is not running its application
}

version = status.status_version_number;
// version number of NEURON CHIP firmware

error_log = status.status_error_log;
// most recent error logged by system

model_number = status.status_model_number;
// model number of NEURON CHIP

```

6

Memory Management

This chapter describes system resources, such as on-chip EEPROM, application buffers, and network buffers, that can be tailored to the needs of a specific application. The following sections discuss how these resources can be reallocated and when you might need to do so.

Reallocating On-Chip EEPROM

Two tables contained in on-chip EEPROM are generated at a default maximum size. These tables are the address table and the domain table. The size of these tables can be reduced through use of compiler options. See the *Compiler Directives* section in Chapter 1 for more information.

If a program doesn't fit into the default memory areas, another alternative when using the NEURON 3150 CHIP is to move parts of the program to other locations in memory. See the *Off-Chip Memory* section later in this chapter.

Address Table

NOTE: See the NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information document (part no. 005-0018-01) for a description of the address table.

The address table contains the list of network addresses to which the node responds. The address table is configured through network management messages from a network management tool.

By default, the address table contains 15 entries. Each address table entry uses five bytes of on-chip EEPROM. The following compiler directive can be used to decrease the number of address table entries:

```
#pragma num_addr_table_entries n
```

n may be any value from 0 to 15

The maximum number of address table entries that a node could require equals the number of connections for that node. Fewer address table entries can be used by sharing address table entries across multiple connections. This capability can only be used if the network management tool used to install the node generates shared entries.

Domain Table

NOTE: See the *NEURON 3120 CHIP* and *NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the domain table.

By default, the domain table is configured for two domains. Each domain uses 15 bytes of on-chip EEPROM. If the application requires only one domain, the size of this table can be reduced through use of one of the following compiler options:

```
#pragma one_domain
#pragma num_domain_entries nn
(nn may be any value from 1 to 2)
```

Allocating Buffers

Pragmas can be used to set certain NEURON CHIP firmware resources such as buffer counts and sizes and receive transaction counts. These values can be set only at compile time. They cannot be configured at run-time. Figure 6-1 illustrates where application and network buffers are used. *Application buffers* are used between the Application and Network processors. *Network buffers* are used between the Network and Media Access Control (MAC) processors.

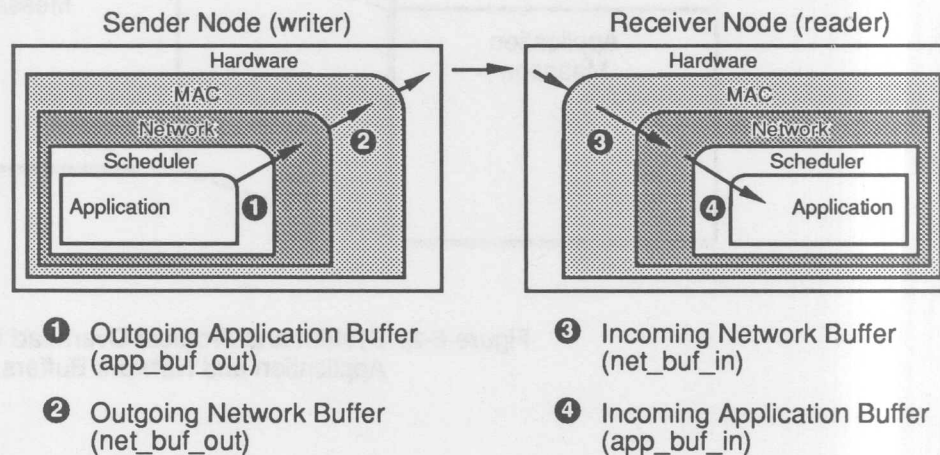


Figure 6-1. Application and Network Buffers

This section outlines a few guidelines for allocating buffers, depending on the needs of an individual application. Also see the *Optimizing LONTALK Response Time Engineering Bulletin* (part no. 005-0011-01) for further information.

Buffer Size

If explicit messaging is used, the buffer size must accommodate the largest message that the application could generate or receive for processing. In some cases, this may require an increase in buffer size. If only network variables are used, buffer sizes are chosen by the compiler.

Figure 6-2 shows the basic components of an application buffer and a network buffer. An application buffer contains application message data and system overhead. A network buffer contains application message data, protocol overhead, and system overhead.

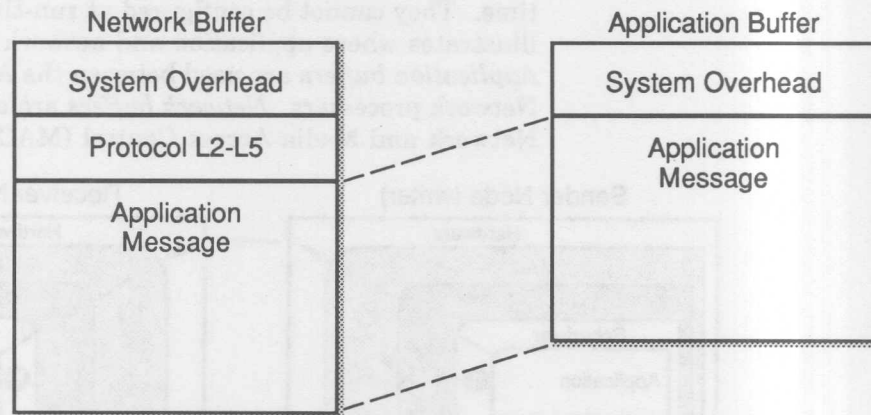


Figure 6-2. System and Protocol Overhead in Application and Network Buffers

Application Buffer Size

The size of an application buffer is equal to:

message_size + 5 bytes of system overhead

If explicit addressing is used, add an additional 11 bytes of system overhead.

For explicit messages, *message_size* equals 1 byte for the message code plus the number of bytes of data. For network variables, *message_size* equals 2 bytes plus the number of bytes in the network variable.

Table 6-1 lists the valid sizes for application buffers. For example, if *message_size* is 40, then you need an application buffer of at least 45 bytes. The next largest valid size for an application buffer is 50 bytes.

Network Buffer Size

The size of a network buffer is less than or equal to:

message_size + 6 bytes of system overhead + 20 bytes of protocol overhead

Protocol overhead ranges from 5 to 20 bytes per message, and this formula uses the maximum. A 40-byte message would thus need a network buffer of at least 66 bytes (see Table 6-1).

Errors

If an input message fits into a network input buffer but does not fit into an application input buffer, the message is discarded. An APP_BUF_TOO_SMALL error is logged. If the message was sent with the acknowledged service, no acknowledgment is sent. If the message was a request, no response is sent.

If an output message fits into an application output buffer but does not fit into a network output buffer, a `NET_BUF_TOO_SMALL` error is logged and the node resets.

Buffer Counts

In most cases, the default number of output buffers is probably sufficient. Increasing the number of buffers on the output side decreases the likelihood of entering preemption mode.

The number of input buffers needed is a function of the type of service used and the type of connections between nodes. If you are using authentication, you may need to increase the number of buffers because authentication doubles the number of messages. If *unicast* connections are used (that is, one node sends a network variable or message to one other node), the default number of buffers is probably sufficient. If *multicast* acknowledged or request connections are used (that is, one node sends a message to multiple nodes), the number of input buffers should be equal to the size of the largest group. If, for example, a node sends a message with the acknowledged or request service to 63 different nodes, the sender node may receive 63 almost simultaneous acknowledgments or responses. The exact number of buffers required is a function of both bit rate and the input clock, so some experimentation may be necessary to determine the optimal number of buffers.

Compiler Directives for Buffer Allocation

The following sections describe the pragmas used for setting the size and number of different types of buffers.

The compiler issues warnings when any of the buffer size pragmas are used and the resulting settings are too small to accommodate all possible network management messages from being properly received or acknowledged.

Outgoing Application Buffers

These pragmas set the size and number of nonpriority and priority buffers between the Application and Network processors for outgoing messages and network variables. See Table 6-1 for a list of default and allowable non-default values.

#pragma app_buf_out_size *n* sets the application buffer size (in bytes) for outgoing priority and nonpriority explicit messages and network variables.

#pragma app_buf_out_count *n* sets the number of application buffers available for outgoing nonpriority explicit messages and network variables.

#pragma app_buf_out_priority_count *n* sets the number of application buffers available for outgoing priority explicit messages and network variables.

Outgoing Network Buffers

These pragmas set the size and number of nonpriority and priority buffers between the Network and MAC processors for outgoing explicit messages and network variables. See Table 6-1 for a list of default and allowable non-default values.

#pragma net_buf_out_size *n* sets the network buffer size (in bytes) for outgoing priority and nonpriority explicit messages and network variables. A size of less than ~~34~~⁴² is not recommended because the node will then be unable to respond correctly to network management messages.

#pragma net_buf_out_count *n* sets the number of network buffers available for outgoing nonpriority messages and network variables.

#pragma net_buf_out_priority_count *n* sets the number of network buffers available for outgoing priority messages and network variables.

These pragmas set the size and number of buffers between the MAC and Network processors for incoming explicit messages and network variables. See Table 6-1 for a list of default and allowable non-default values.

<code>#pragma net_buf_in_size <i>n</i></code>	sets the network buffer size (in bytes) for incoming explicit messages and network variables. A size of less than 50 is not recommended because the node will then be unable to respond correctly to network management messages.
<code>#pragma net_buf_in_count <i>n</i></code>	sets the number of network buffers available for incoming explicit messages and network variables.

Incoming Application Buffers

These pragmas set the size and number of buffers between the Network and Application processors for incoming explicit messages and network variables. See Table 6-1 for a list of default and allowable non-default values.

<code>#pragma app_buf_in_size <i>n</i></code>	sets the application buffer size (in bytes) for incoming explicit messages and network variables. A size of less than 22 is not recommended because the node will then be unable to respond correctly to network management messages.
---	---

#pragma app_buf_in_count *n* sets the number of application buffers available for incoming explicit messages and network variables.

Number of Receive Transactions

The number of incoming transactions that can be handled concurrently by the Network processor is determined by the receive transaction array. This pragma sets the number of entries in the array. See Table 6-1 for a list of default and allowable nondefault values. The size of a receive transaction entry is 13 bytes.

#pragma receive_trans_count *n* sets the number of entries in the receive transaction array.

Table 6-1. Values for Buffer Sizes and Counts (Part 1 of 3)

Pragma	Values Allowed	Default
app_buf_out_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes	A

■ Pages 6-11, 6-12, and 6-13.

Replace the existing table with the table below:

Table 6-1. Values for Buffer Sizes and Counts (Part 1 of 3)

Pragma	Values Allowed	Default
app_buf_out_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes	A
app_buf_out_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	E
app_buf_out_priority_count	0, 1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	E
net_buf_out_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes (Note: a value of less than 42 is not recommended-see preceding pages for explanation.)	B
net_buf_out_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	E
net_buf_out_priority_count	0, 1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	E
net_buf_in_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes (Note: a value of less than 50 is not recommended-see preceding pages for explanation.)	66
net_buf_in_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	2
app_buf_in_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, or 255 bytes (Note: a value of less than 22 is not recommended-see preceding pages for explanation.)	C
app_buf_in_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, or 63	2
receive_trans_count	1..16	D

Table 6-1. Values for Buffer Sizes and Counts (Part 2 of 3)

A. app_buf_out_size default

If outgoing messages are sent with `msg_send()`:

If explicit addressing is used:
 $A = 66$

If explicit addressing is not used:
 $A = 50$

If outgoing explicit messages are not sent, and:

If explicit addressing is used for network variables:
 $A = \max(34, 19 + \text{sizeof}(\text{largest output NV}))$

If explicit addressing is not used:
 $A = \max(20, 8 + \text{sizeof}(\text{largest output NV}))$

B. net_buf_out_size default

If outgoing explicit messages are sent with `msg_send()` or `resp_send()`:
 $B = 66$

else:
 $B = \max(42, 22 + \text{sizeof}(\text{largest NV}))$

C. app_buf_in_size default

If any explicit message functions or events are used (incoming or outgoing):

If explicit addressing is used:
 $C = 66$

If explicit addressing is not used:
 $C = 50$

If no explicit message functions or events are used, and:

If explicit addressing is used for network variables:
 $C = \max(34, 19 + \text{sizeof}(\text{largest NV}))$

If explicit addressing is not used:
 $C = \max(22, 8 + \text{sizeof}(\text{largest NV}))$

D. receive_trans_count default

If explicit messages are used:

$$D = \max(8, \min(16, \# \text{ input NVs} + 2))$$

Table 6-1. Values for Buffer Sizes and Counts (Part 3 of 3)

D. receive_trans_count default

If explicit messages are received by the application program:

$$D = \max(8, \min(16, \# \text{ of non-config input NVs} + 2))$$

If explicit messages are not received by the application program:

$$D = \min(16, \# \text{ of non-config input NVs} + 2)$$

E. app_buf_out_count, app_buf_out_priority_count,
net_buf_out_count, and net_buf_out_priority_count defaults

If the application is linked for a NEURON 3150 CHIP:

$$E = 2$$

If the application is linked for a NEURON 3120 CHIP:

$$E = 1$$

Note: Explicit addressing is used by a program if it references any of the following:

msg_in.addr
resp_in.addr
msg_out.dest_addr
nv_in_addr

Using NEURON CHIP Memory

On-chip memory on the NEURON 3150 CHIP consists of RAM and EEPROM.

Off-chip memory on the NEURON 3150 CHIP consists of ROM and optionally RAM and/or EEPROM. You specify the starting page number and the number of pages (a page is 256 bytes) when the node is defined. See the *LONBUILDER User's Guide, Chapter 6*. The start of ROM is fixed at 0000. The regions of memory must be in the order shown in Figure 6-3. They need not be contiguous, but they cannot overlap.

Memory mapped I/O devices can be connected to the NEURON 3150 CHIP. The devices should respond only to memory addresses which correspond to the shaded areas in Figure 6-3, below.

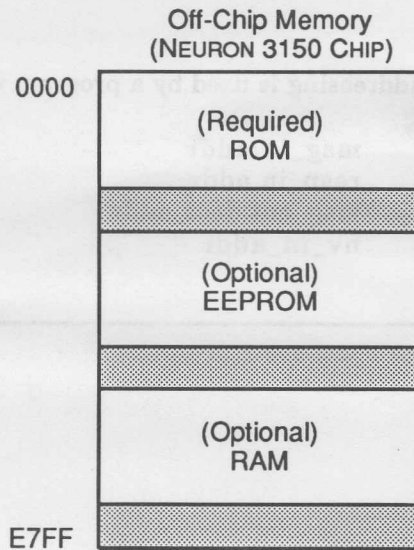


Figure 6-3. Off-Chip Memory on the NEURON 3150 CHIP

On-chip memory on the NEURON 3120 CHIP consists of ROM, RAM, and EEPROM. The NEURON 3120 CHIP does

not permit off-chip memory. Figure 6-4 summarizes the NEURON 3120 and 3150 CHIP memory maps.

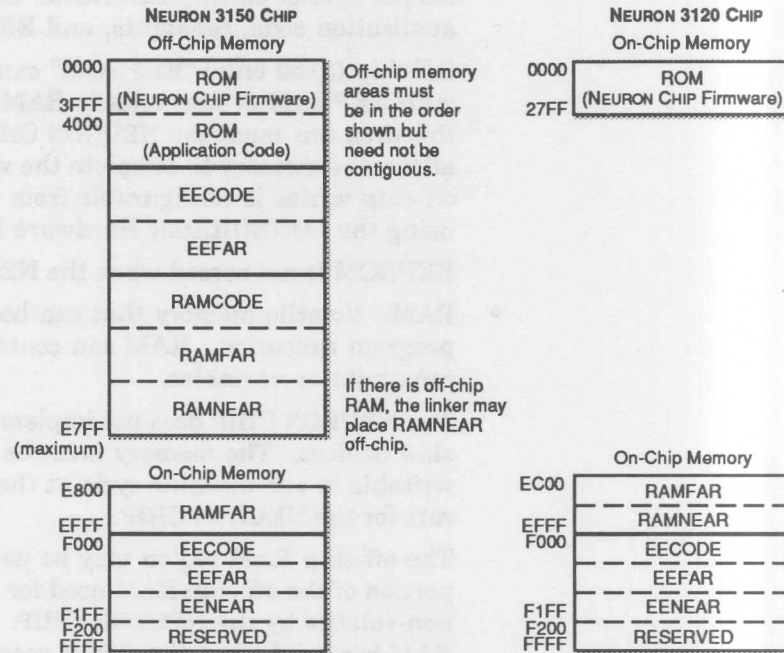


Figure 6-4. Memory Maps for NEURON 3150 CHIP and NEURON 3120 CHIP, Showing Areas Defined by the Linker

Memory Regions

The definitions of the three memory regions are:

- **ROM:** Non-volatile memory initialized before program execution on a node. ROM cannot be changed by the program. It is used for the NEURON CHIP firmware and can optionally (3150 only) contain application code and constants.

Off-chip (3150 only) "ROM" may be implemented with any non-volatile memory technology, including ROM, PROM, EPROM, EEPROM, or non-volatile RAM.

- **EEPROM:** Non-volatile memory that can be changed during program execution. Memory writes require 20 ms per byte of on-chip EEPROM. EEPROM can contain application code, constants, and EEPROM variables. Off-chip (3150 only) "EEPROM" can be implemented with EEPROM or non-volatile RAM. Memory writes to this area can cause the NEURON CHIP to delay. This allows the memory to complete the write. The delay for off-chip writes is configurable from 0 to 255 milliseconds using the LONBUILDER Hardware Properties screen. EEPROM is not zeroed when the NEURON CHIP is reset.
- **RAM:** Volatile memory that can be changed during program execution. RAM can contain application code, constants or variables.

The NEURON CHIP does not implement wait states for slow devices. The memory must be readable and writable in one machine cycle at the selected input clock rate for the NEURON CHIP.

The off-chip RAM region may be used for code. Any portion of the off-chip RAM used for code is assumed to be non-volatile by the NEURON CHIP. The remainder of RAM is zeroed when the chip is reset.

Memory Areas

The three regions are further divided into memory areas as follows:

- The ROM region has a system area and a user area (NEURON 3150 CHIP only). The system area is 16 Kbytes on a NEURON 3150 CHIP and 10 Kbytes on a NEURON 3120 CHIP. The user area is also named ROM. The NEURON C compiler and linker place executable code and constant data in this area.
- The EEPROM region consists of three areas:

EECODE
EEFAR
EENEAR

If there is both on-chip and off-chip EEPROM, each region of EEPROM will have its own section of EECODE and EEFAR. There is only one section of EENEAR, and it is always located on-chip. All of these are user areas.

EECODE contains executable code and constant data. The `eprom` keyword in NEURON C forces the compiler to place a specific object in this area.

The EEFAR area contains variables declared with the `far` keyword combined with either the `config` or `eprom` keywords.

The EENEAR area contains variables declared with either the `config` or `eprom` keywords. It is the default, but is limited to a total size of 255 bytes.

- The RAM region consists of three areas:

RAMCODE

RAMFAR

RAMNEAR

RAMCODE can only be located off-chip (NEURON 3150 CHIP only). It contains executable code and constant data. By using the `ram` keyword in a declaration, the user can explicitly direct executable code and constant data to this area.

RAMFAR may be located both on-chip and off-chip. There may be one or two sections of RAMFAR in the on-chip RAM. If there is off-chip RAM, it may contain only one RAMFAR area. The RAMFAR area contains variables.

There may be only one RAMNEAR area. It may be located on-chip (NEURON 3150 and 3120 CHIPS) or off-chip (NEURON 3150 CHIP only). The location is automatically determined by the linker. The RAMNEAR area contains variables and is the default. It is generally limited to a total size of 256 bytes.

However, the maximum allowable size may be smaller under certain circumstances depending on the amount of memory the user has allocated for buffers. See the *Compiler Directives for Buffer Allocation* section earlier in this chapter.

Also see the *Special Keywords for Non-Default Memory Usage* section later in this chapter.

Default Memory Usage

If no special keywords are included in the declarations of variables or functions or other pieces of a NEURON C program, the pieces of the program are located in memory by the linker using the following rules.

All executable code objects (functions, when clauses, tasks) as well as string constants and data declared as `const` are placed in the ROM or EECODE areas. The linker places these objects wherever they fit. For the NEURON 3150 CHIP, the linker first tries to put an object in the user area of *off-chip* ROM. If the object doesn't fit in ROM, the linker attempts to put it in the *off-chip* EECODE area of memory. Finally, the linker will attempt to put the object in the *on-chip* EECODE area of memory. (On the NEURON 3120 CHIP, *only* the on-chip EECODE area of memory is available to the linker.)

Data objects declared with either the `config` or `eeprom` keywords are placed in the on-chip EENEAR area of memory. The linker places all other data objects in the RAMNEAR area of memory.

The linker's placement of variables and functions can be modified using the special keywords described in the next section.

Special Keywords for Non-Default Memory Usage

If you receive an error message at link time that part of your program doesn't fit into the available default memory, you can change the declarations of variables or functions using special NEURON C keywords. These keywords enable you to move the variables or functions to other locations in memory. The special keywords `far`, `ram`, and `eeeprom` are described below.

Memory reads and writes (by the application program) to data in the EENEAR and RAMNEAR areas take advantage of special addressing modes in the NEURON CHIP that generate more efficient code (fewer bytes per instruction and fewer cycles per read or write).

far Keyword

When data objects do not fit into the RAMNEAR area of memory, the following messages appear:

```
Error: No more memory in RAMNEAR area
Error: Could not relocate segment in file '<program>.no'
```

You can direct the linker to put the objects into the RAMFAR area of memory by including the `far` keyword in the NEURON C data declaration. For example:

```
far int varname;
```

Similarly, when `config` or `eeeprom` objects do not fit into the EENEAR area of memory, the following messages appear:

```
Error: No more memory in EENEAR area
Error: Could not relocate segment in file '<program>.no'
```

You can direct the linker to put the objects into the EEFAR area of memory by also including the `far` keyword in the NEURON C data declaration. For example:

```
far eeeprom int varname;
```

A data table too large to fit into the EENEAR area could thus be redirected to the EEFAR area of memory.

As a general guideline, leave data that is *more* frequently used in the NEAR areas of memory if possible. Use of the NEAR areas generates relatively smaller instructions (which do not take as long to execute) than use of the FAR areas.

ram Keyword (for Functions)

Functions can be redirected to the RAMCODE area of memory by including the ram keyword in the NEURON C function definition. The RAMCODE area is only available in off-chip RAM memory attached to a NEURON 3150 CHIP. The RAM must be non-volatile (for example, battery-backed), otherwise the node may fail after power up.

The ram keyword can go anywhere before the function name. For example:

```
ram int fn() { ... statements }
```

The ram keyword is useful for functions that a network management tool may change frequently after installation.

eeeprom Keyword

On the NEURON 3150 CHIP, functions are located in ROM by default. However, functions can also be redirected to the EECODE area of memory by including the eeeprom keyword in the function definition. For example:

```
eeeprom int fn() { ... statements }
```

The eeeprom keyword is useful for functions that may be changed infrequently after installation by a network management tool.

This keyword also allows the application program to indicate variables whose values are preserved across power outages by locating the variables in EENEAR rather than in RAMNEAR. In general, eeeprom variables have a limited capability to accept changes.

Data objects can be redirected to the EENEAR area of memory by including the `eeeprom` keyword in the declaration, as described earlier. For example:

```
eeeprom int varname;
```

Data objects can be redirected to the EEFAR area of memory by including the `eeeprom` *and* `far` keywords in the declaration, as described earlier. For example:

```
eeeprom far int varname;
```

Network variables are directed to the EENEAR area with either the `eeeprom` or `config` keyword. The `far` keyword may also be used with network variables similar to the example above.

Initializers for `eeeprom` class variables take effect when the application image is loaded from an external system, such as the LONBUILDER Developer's Workbench or a network management tool. Reloading a program has the effect of reinitializing all `eeeprom` variables. Restarting a node or powering it up does *not* re-initialize the `eeeprom` variables.

Writing a value in on-chip EEPROM takes approximately 20 ms before the value takes effect. If this write time is cut short, the value may not have been written or, if written, the value may not be non-volatile. (Examples of cases when the write time might be shortened are when the node powers down due to a power outage, when the node is externally reset, or when a watchdog timeout occurs and the node is reset.)

When the Program Is Relinked

The compiler directs the application code to the proper areas of memory. The linker assigns memory locations. Invoking the compiler and linker is handled automatically by the LONBUILDER Project Manager. If you change the memory map, the project manager will only relink the program, not recompile it, since only the memory locations have changed.

Memory Use

This section outlines the amount of memory used by certain elements in your program. For a description of the actual memory used by your program, see the link summary produced by the project manager.

RAM Use

RAM is used as follows:

- Each timer object uses 4 bytes.
- Each `io_changes` event (any type) uses 3 bytes.
- The following amounts pertain to global and static data as declared in the program (except for `EEPROM` and `config` variables). These amounts also apply to network variables.

<code>char</code>	1 byte
<code>int</code>	1 byte
<code>enum</code>	1 byte
<code>long</code>	2 bytes

structures	Sum of the size of the elements. Each 8 bits (or fraction of 8 bits) of consecutive bitfields uses up a byte. No bitfield can span a byte boundary. No word alignment is performed.
------------	---

unions	Size of the largest element
--------	-----------------------------

message tag	0
-------------	---

I/O object	0
------------	---

code	Size of code (for functions declared with <code>ram</code> keyword)
------	---

EEPROM Use

Approximately 65 bytes of EEPROM is used as constant system overhead. In addition, EEPROM is used as follows:

- A domain requires 15 bytes for configurable information, to define the domain address, subnet number, node number, and authentication key. A system can have a maximum of two domains and must have at least one domain. The default is two domains. See *Domain Table* section earlier in this chapter.
- Each address table entry requires 5 bytes. A maximum of 15 address table entries are allowed. The minimum is 0. The default is 15 entries. See *Address Table* section earlier in this chapter.
- Each network variable declared (input or output) uses 6 bytes for configuration and control information. If you use the SNVT Self-Identification (SI) feature, there is an additional 4-byte fixed overhead plus 2 additional bytes per network variable (minimum).
- Variables declared as `eeeprom` and `config` in your program use the corresponding amount of EEPROM.
- when clause tables are placed in CODE (EEPROM on a NEURON 3120 CHIP). Each when clause results in an entry from 3 to 6 bytes (most are 3 bytes). This code space is usually slightly smaller than the equivalent code generated by an `if` statement. Additional code space may result from when clauses containing user-defined events.

Usage Tip for Memory Mapped I/O

Memory mapped IO devices can be attached to the NEURON 3150 CHIP. These devices should respond only to memory addresses which are outside the configured memory map areas for ROM, EEPROM, and RAM. The LONBUILDER Hardware Properties screen (see the *LONBUILDER User's Guide, Chapter 6*) allows configuration of the memory map.

A convenient method of access to memory mapped I/O from a NEURON C program is to declare a constant pointer to the

block of control addresses for the device. In the following example, a hypothetical memory mapped I/O device has two control registers and a 16-bit data register, at addresses x , $x+1$, $x+2$, and $x+3$, respectively. The device is connected to respond to the NEURON 3150 CHIP addresses of 0x8800 to 0x8803. The fragment of NEURON C code shown below is a good way to access the device.

```
typedef struct {
    unsigned short int    controlReg1;
    unsigned short int    controlReg2;
    unsigned long int     dataReg;
} *PMemMapDev;

const PMemMapDev pDevice = (PMemMapDev) 0x8800;

// Read from device ...
unsigned int x, y;
unsigned long z;

x = pDevice->controlReg1;
y = pDevice->controlReg2;
z = pDevice->dataReg;

// Write to device ...
unsigned int x, y;
unsigned long z;

pDevice->controlReg1 = x;
pDevice->controlReg2 = y;
pDevice->dataReg = z;
```

What to Try When a Program Doesn't Fit on a NEURON 3120 CHIP

The following discussion contains tips and techniques for reducing the EEPROM requirements of a program for purposes of getting it to fit on a NEURON 3120 CHIP. It is expected that the techniques below would be attempted in roughly the order presented.

The Link Summary contains information on a program's current memory usage. The summary information includes an estimate of the additional memory required. The link summary is optionally output to the BUILD.LOG

file, and is also included in the optionally-produced link map file. Either or both of these options can be selected from the LONBUILDER IDE menu "Options/Project". For more information, see the *LONBUILDER User's Guide, Chapter 4*.

Reduce the Number of Address Table Entries

The minimum number of address table entries that a fully connected NEURON application program can use is the sum of the number of output network variables and bindable message tags. (A bindable message tag is one which does *not* include `bind_info` (nonbind) in its declaration.) For example, an application with one message tag and two output network variables (one of which is an array of four elements), would need a minimum of six address table entries.

However, additional address table entries may be needed for input network variables which are in one or more groups, one entry being used for each group. Finally, each connection to a node's "msg_in" tag will use an address table entry.

If your program does not explicitly receive messages (and therefore will have no connections to the "msg_in" tag), and it has only a few network variables which will each be connected only in a point-to-point manner (i.e. no group connections), you could easily reduce the number of address table entries. Note, however, that a network management tool could make connections to "msg_in" in any case. Other situations could require further analysis to determine if the number of address table entries could be reduced.

The default number of address table entries is 15. The value can be reduced with the `#pragma num_addr_table_entries` (see the *Compiler Directives* section in Chapter 1). Reducing the number of address table entries will save 5 bytes of EEPROM per entry eliminated.

Remove Self-Identification Data if Not Needed

The NEURON C compiler places self-identification data in the node's program space. On the NEURON 3120 CHIP, this consumes EEPROM. If your program is not using SNVTs, you can consider removing the self-identification data. This can be done by specifying the following compiler directive:

```
#pragma disable_snvt_si
```

Take Advantage of NEURON CHIP Firmware Default Initialization Actions

The NEURON CHIP firmware automatically sets all RAM variables to zero each time the chip resets, and also when the NEURON C function `application_restart()` is called. After this action, the NEURON C application program is launched. The first action of a NEURON C application program is to execute code to initialize any RAM variables to non-zero values. Then, if a task associated with the `when(reset)` clause exists, it is called.

Therefore, use of compile-time initializers to set RAM variables to zero is "free". Eliminate any code in the `when(reset)` clause's task which is used to set RAM variables to zero, as it is unnecessary.

Also, compile-time initializers of I/O output objects are "free". This is true regardless of initializer value. The use of compile-time initializers for I/O will use less code space than corresponding calls to `io_out()` in the `when(reset)` clause's task.

Finally, at reset time, the NEURON C application timers are all turned off automatically. Eliminate any code in the `when(reset)` clause's task which explicitly turns off application timers.

Use NEURON C Utility Functions Effectively

There are several NEURON C utility functions which can be used to reduce code requirements. For example, there are `min()`, `max()`, and `abs()` functions, as well as other utility functions which may be used for common operations. Use of these functions will generally be more code-space efficient than coding the operations in-line using C operators.

The NEURON C functions also include such utilities as the `timers_off()` function. This function turns off all application timers with a single function call. This function call takes less space than the corresponding assignment of zero to a single timer, although it takes longer to execute. Thus, if your program contains a single application timer, and you turn it off by assigning zero to it, consider using this function instead in order to save code

■ Page 6-27

Add the follont) {

For exampl
value in th

<type> fir

(which appears in
declared first, then

When this fi
compiler), thuses the rules of
shown: interpreted as a
ceed the bounds of
<type>Increase the code
<type>
<type>
<type>generate better code
are observed.

In addition t
unsigned shoeler if it is a signed
implies that
efficient that
of the most e
instruction s
SI Standard C for
integers. g this technique

Add the followd for array
n of such an array.
ie fastaccess
Observe D to all arrays in
s al and a2 to both

The order of
an effect on c
declared on tl
NEURON C on syntax,
on the stack, ys may appear on
accesses (load, hat the fastaccess
are generally iter.

linker will locate
(A NEURON CHIP
y global arrays as
fragmentation.

ibed in Appendix C.

is which are placed in the
IP (see the table at the
endix C for a complete list of
possible, avoid use of such
structures which cause use of

generate more compact code
tions on them more closely
architecture and instruction
es to be locals rather than
long, and to be unsigned

SEE INSERT

Eliminate Common Sub-Expressions

The NEURON C compiler does not automatically eliminate common sub-expressions. Performing this optimization by hand would, in most cases, reduce code size. Consider the following "before-and-after" example, which saves 4 bytes of code:

Before:

```
int a, b, c, d, e;
void f(void) {
    d = (a * 2) + (b * c * 4);
    e = a - (b * c * 4);
}
```

After:

```
int a, b, c, d, e;
void f(void) {
    int temp;
    temp = b * c * 4;
    d = (a * 2) + temp;
    e = a - temp;
}
```

Use Function Calls Liberally

Since function calls are relatively cheap in terms of the code space and execution time overhead, replacing even a single line of complex code with an equivalent function may reduce code space if that line of code is used two or more times in a program. Some lines of code involving network variables may not look complex, but the underlying operations may be.

For example, consider the increment of an element in a structure which was part of an array of network variables;

this operation would generate a considerable amount of code. Replacing two such occurrences with a single function call would save code space at the expense of a minor performance penalty.

Also, consider passing expressions and values as function actual parameters, rather than using global variables. Accesses to parameters are generally more efficient than (or are no worse than) accesses to globals.

Use the Alternate Initialization Sequence

Use of the `#pragma disable_mult_module_init` will save 2 or 3 bytes of EEPROM code space. This pragma specifies to the compiler to generate any required initialization code directly in the special init and event block, rather than as a separate procedure callable from the special init and event block.

The in-line method, which is selected as a result of use of this pragma, is more efficient in memory usage (it typically saves 3 bytes if initialization code is present, and saves 2 bytes if no initialization code is present). However, the drawbacks of using the pragma are: (1) the in-line initialization area is limited in length, and (2) there can be no linkage from the program's initialization code to application library or custom image initialization code (this is typically not a problem for a NEURON 3120 CHIP).

Reduce the Number of Domains

If it is known that the application node will always be a member of only one domain, then either the `#pragma num_domains 1` directive or the `#pragma one_domain` directive can be used to save 15 bytes of EEPROM. The default number of domain entries is 2, and each domain entry uses 15 bytes.

Use C Operators Effectively

The ANSI C language has a rich set of operators. Using them effectively can produce very efficient code.

For example, use of the C `? :` operator rather than use of an `if - else` statement for alternative assignments to the same left-hand-side may reduce code space, especially if the left-hand side expression is complex.

Use NEURON C Extensions Effectively

The NEURON C language contains features which exist primarily to help write efficient code.

For example, if a program had two input network variables, and had a single task executed when either variable was updated, it would be more efficient to code it as shown in the "after" example, below. Likewise, use of a single `when` clause with the `nv_update_occurs` event referencing just an array name would be more efficient than using multiple `when` clauses, one for each element of an array.

Before:

```
when (nv_update_occurs(var1))
when (nv_update_occurs(var2))
{
    // task ...
}
```

After:

```
when (nv_update_occurs)
    // Use "unqualified" event to cover
    // all vars
{
    // task ...
}
```

System Library on a NEURON 3120 CHIP

On a NEURON 3120 CHIP, all application code is placed in on-chip EEPROM. In addition, when any of the following functions are used, they are brought in from a system library and placed in on-chip EEPROM.

The system library functions for the NEURON 3120 CHIP are:

<code>access_address()</code>	
<code>access_domain()</code>	
<code>access_nv()</code>	
<code>bcd2bin()</code>	
<code>bin2bcd()</code>	
<code>flush_wait()</code>	
<code>go_unconfigured()</code>	
<code>io_edgelog_preload()</code>	
<code>io_preserve_input()</code>	
<code>muldiv()</code>	
<code>muldivs()</code>	
<code>power_up()</code>	
<code>retrieve_status()</code>	
<code>reverse()</code>	
<code>update_address()</code>	
<code>update_clone_domain()</code>	
<code>update_config_data()</code>	
<code>update_domain()</code>	
<code>update_nv()</code>	
<code>_bitf_sign_ext()</code>	(any use of a signed bitfield)
<code>_dualslope_input()</code>	(any <code>io_in()</code> for a dualslope object)
<code>_dualslope_start()</code>	(any <code>io_in_request()</code> for a dualslope object)
<code>_edgelog_input()</code>	(any <code>io_in()</code> for an edgelog object)
<code>_io_set_clock_x2()</code>	(any <code>io_set_clock()</code> for an edgelog object)
<code>_ir_input()</code>	(any <code>io_in()</code> for an infrared object)
<code>_magcard_input()</code>	(any <code>io_in()</code> for a magcard object)
<code>_memcpy16()</code>	(<code>memcpy()</code> or struct assign length ≥ 256)
<code>_memset16()</code>	(<code>memset()</code> with length ≥ 256)
<code>_msg_data_blockget()</code>	(<code>memcpy()</code> out of <code>msg_in.data</code>)
<code>_msg_in_addr_ptr()</code>	(any use of <code>msg_in.addr</code>)
<code>_msg_out_addr_ptr()</code>	(any use of <code>msg_out.dest_addr</code>)
<code>_muxbus_read()</code>	(any <code>io_in()</code> for a muxbus object)
<code>_muxbus_reread()</code>	(any <code>io_in()</code> for a muxbus object)
<code>_muxbus_rewrite()</code>	(any <code>io_out()</code> for a muxbus object)
<code>_muxbus_write()</code>	(any <code>io_out()</code> for a muxbus object)
<code>_neurowire_slave()</code>	(any <code>io_in()</code> or <code>io_out()</code> for a neurowire slave object)
<code>_resp_data_blockset()</code>	(<code>memcpy()</code> into <code>resp_out.data</code>)
<code>_resp_in_addr_ptr()</code>	(any use of <code>resp_in.addr</code>)

See Section C.2 in Appendix C for a discussion of each function.

An application linked for a NEURON 3120 CHIP using any of the above functions (including an emulator or SBC emulating a NEURON 3120 CHIP) will require more on-chip EEPROM than the same application linked for a NEURON 3150 CHIP. This is because on the NEURON 3120 CHIP these functions are located in a system library instead of in the NEURON CHIP firmware. Examination of the link map can provide a measure of the EEPROM memory used by these functions. To obtain an estimate of the NEURON 3120 CHIP EEPROM required for these functions, follow these steps:

- 1 Select the Generate Link Map option under *Configuring LONBUILDER Project Parameters* in the project configuration screen described in Chapter 4 of the *LONBUILDER User's Guide*.
- 2 Create a hardware properties object with a NEURON CHIP type of 3120 as described under *Defining Properties* in Chapter 6 of the *LONBUILDER User's Guide*.
- 3 Define an emulator or SBC node with the properties defined in step 2.
- 4 Select the Automatic Build command in the project pull-down menu (or press ctrl-F10).

After the build in step 4 is completed, the link map for the node defined in step 3 will contain the NEURON 3120 CHIP EEPROM requirements for the system library functions. See the *LONBUILDER User's Guide, Chapter 4 and Appendix C* for more information on the link map.

Appendix A

Sample Application

This appendix contains node definitions and programs for the sample automobile application introduced in Chapter 3. The following files are included:

- door.nc
- interior.nc
- key.nc
- hswitch.nc
- headlight.nc

Introduction

This sample application contains seven nodes for an auto control system running five different programs, as follows:

<i>Node Name</i>	<i>Program Name</i>
fl_door	door
fr_door	door
hswitch	hswitch
interior_light	interior
key	key
l_head	headlight
r_head	headlight

The programs for the left door and right door are the same. The left door lock is configured as a master lock that can activate the right door lock, by connecting the lock output network variable at the left door to the lock input network variable at the right door, but not vice versa.

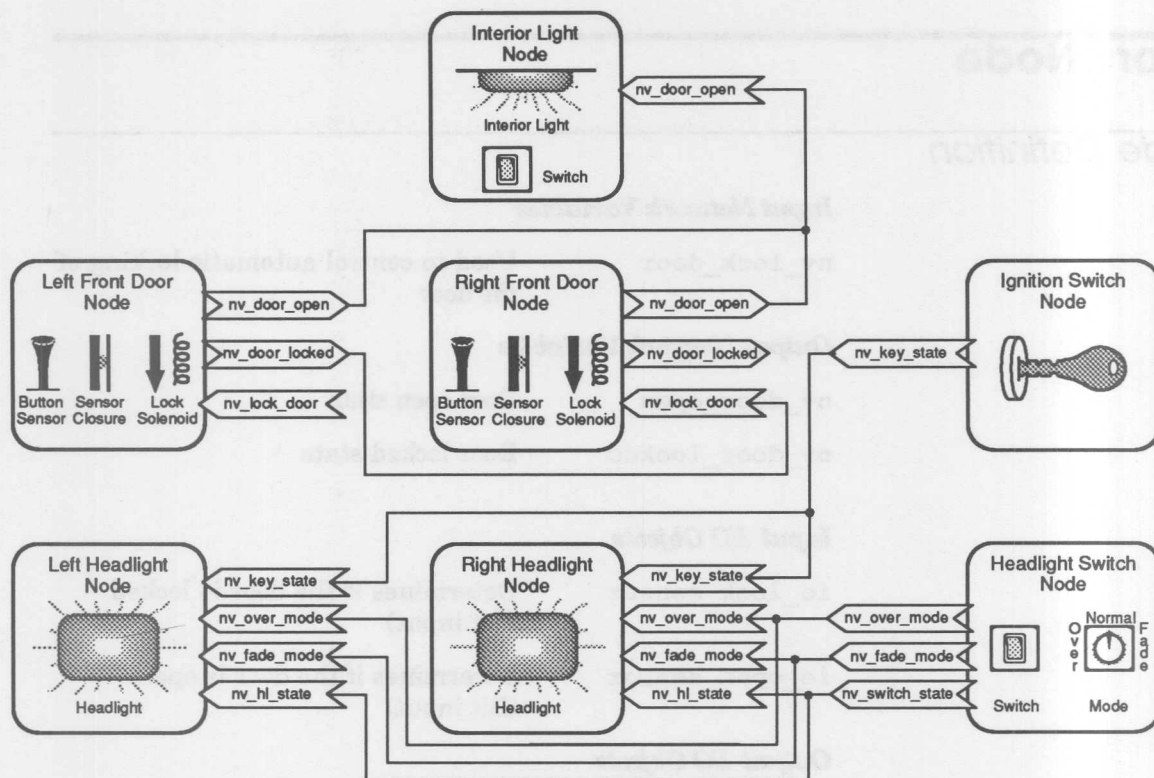


Figure A-1. Sample Auto Control System

The interior light program may be used in a four-door car, but in this example, it is used in a two-door car.

Door Node

Node Definition

Input Network Variables

<code>nv_lock_door</code>	Used to control automatic locking of the door
---------------------------	---

Output Network Variables

<code>nv_door_open</code>	Door open state
<code>nv_door_locked</code>	Door locked state

Input I/O Objects

<code>io_lock_sensor</code>	Determines if the door is locked (Bit input)
<code>io_open_sensor</code>	Determines if the door is open (Bit input)

Output I/O Objects

<code>io_lock_control</code>	Used to cause the door to lock and unlock - solenoid activating lock (Bit output)
------------------------------	---

Functional Specification

- Initially the door is closed and unlocked.
- When the door opens, the output network variable `nv_door_open` is set to `ST_ON`.
- When the door closes, the output network variable `nv_door_open` is set to `ST_OFF`.
- When the door is locked:

The output network variable `nv_door_locked` is set to `ST_ON`.

The io_lock_control solenoid is activated.

- When the door is unlocked, the output network variable nv_door_locked is set to ST_OFF.
- When the door is told to lock, the io_lock_control solenoid is activated.
- When the door is told to unlock, the io_lock_control solenoid is deactivated.

door.nc

```
// Purpose: Example program. This application
// controls a car door. A lock input is used to
// provide automatic locking of the doors. A
// "lock state" variable is provided to indicate
// that this door is locked (using these
// together, one door can lock all the doors in
// the car).

#include <io_types.h>
#include <snvt_lev.h>

// Network variable declarations
network output SNVT_lev_disc nv_door_open;
// ST_ON if door is open
network output SNVT_lev_disc nv_door_locked;
// ST_ON if door is locked
network input SNVT_lev_disc nv_lock_door =
    ST_OFF;
// ST_ON to cause door to lock

// I/O object declarations
const bit_t DOOR_LOCK_COMMAND = 0;
const bit_t DOOR_UNLOCK_COMMAND = 1;
IO_0 output bit io_lock_control = 0;
// Used to control
// automatic lock

const bit_t DOOR_UNLOCKED_STATE = 0;
const bit_t DOOR_LOCKED_STATE = 1;
IO_7 input bit io_lock_sensor;
// Door lock sensor

const bit_t DOOR_OPEN_STATE = 0;
const bit_t DOOR_CLOSED_STATE = 1;
IO_9 input bit io_open_sensor;
// Door opened sensor

// Function prototype
void set_lock(boolean lock_door);
```

```

// Node reset task
when (reset)
{
    // Get the current door open state
    nv_door_open = (io_in(io_open_sensor)
                    == DOOR_OPEN_STATE) ? ST_ON : ST_OFF;

    // Get the current door lock state
    set_lock(io_in(io_lock_sensor) ==
            DOOR_LOCKED_STATE);

    io_change_init(io_open_sensor);
    io_change_init(io_lock_sensor);
}

// Door open/close task
when (io_changes(io_open_sensor))
{
    nv_door_open = (input_value ==
                    DOOR_OPEN_STATE) ? ST_ON : ST_OFF;
}

// Door locked task
when (io_changes(io_lock_sensor))
{
    set_lock(input_value == DOOR_LOCKED_STATE);
}

// nv_lock_door update task
when (nv_update_occurs(nv_lock_door))
{
    set_lock(nv_lock_door == ST_ON);
}

```

```
// Function to lock or unlock the door
void set_lock(boolean lock_door)
{
    if (lock_door) {
        io_out(io_lock_control, DOOR_LOCK_COMMAND);
        nv_door_locked = ST_ON;
    } else {
        io_out(io_lock_control, DOOR_UNLOCK_COMMAND);
        nv_door_locked = ST_OFF;
    }
}
```

Interior Light Node

Node Definition

Input Network Variables

nv_door_open	Door open state
--------------	-----------------

Input I/O Objects

io_on_switch	Manual on/off switch (Bit input)
--------------	----------------------------------

Output I/O Objects

io_int_light	Interior light brightness (Pulsewidth output)
--------------	--

Functional Specification

- When any of the doors open, the interior light turns on at full brightness.
- When the last door closes, the light slowly dims off.
- When the manual on/off switch is turned on, the interior light turns on at full brightness.
- When the manual on/off switch is turned off:

If all the doors are closed the interior light goes out immediately.

If any of the doors are open, the interior light stays on (and slowly dims when the last door is closed).

interior.nc

```
// Purpose: Example program. This application
// is the interior light for a car. Any number
// of doors can be connected. When any of the
// doors is opened, the light turns on. When
// all doors close, the light slowly fades out.
// A manual on/off switch is also provided.

#include <io_types.h>
#include <snvt_lev.h>

// Network variable declarations
network output SNVT_lev_disc nv_light_on;
// ST_ON if light is on
network input SNVT_lev_disc sync nv_door_open;
// ST_ON if a door opens
// Use a synchronous NV
// so that all doors may
// be polled at once
```

```

// I/O object declarations
const pulsewidth_t MIN_BRIGHTNESS = 0;
const pulsewidth_t MAX_BRIGHTNESS = 255;
IO_0 output pulsewidth invert clock (7)
    io_int_light = 0;
                                // Interior light

const bit_t SWITCH_OFF_STATE = 1;
const bit_t SWITCH_ON_STATE = 0;
IO_7 input bit io_on_switch;
                                // Manual on/off switch

// Timer declarations
mtimer dimmer_timer;    // Dimmer control sampling timer

// Global variable declarations
unsigned short int light_brightness;
                                // Brightness of the
                                // light
boolean manual_on;    // Manual on/off switch
                                // state
short int num_doors_open = 0;
                                // Number of open doors

const short int bright_step = 5;
                                // Brightness increment
const unsigned long int timer_period = 50;
                                // Period for dimming
                                // lights

// Function prototype
void set_interior(boolean light_on);

```

```

// Node reset task
when (reset)
{
    // Get the on/off switch state
    manual_on = (io_in(io_on_switch) == SWITCH_ON_STATE);
    set_interior(manual_on);

    io_out(io_int_light, light_brightness);
    io_change_init(io_on_switch);

    // Get the state of all connected doors
    poll(nv_door_open);
}

// Manual on/off switch on task
when (io_changes(io_on_switch))
{
    // Get the new switch state
    manual_on = (input_value == SWITCH_ON_STATE);

    // If the switch is now on, turn the light
    // on. If the switch is now off, only turn
    // the light off if all the doors are closed.
    if (manual_on) {
        io_out(io_int_light, light_brightness);
        set_interior(TRUE);
        dimmer_timer = 0;
    } else if (num_doors_open == 0) {
        io_out(io_int_light, light_brightness);
        set_interior(FALSE);
    }
}

```

```

// Door changing state task
when (nv_update_occurs(nv_door_open))
{
    // If door open increment num_doors_open, else decrement
    num_doors_open += (nv_door_open == ST_ON) ? 1 : -1;

    if ((num_doors_open > 0) || manual_on) {
        // Turn the light on
        io_out(io_int_light, light_brightness);
        set_interior(TRUE);
        dimmer_timer = 0;
    } else
        dimmer_timer = timer_period; // Start the dimmer
}

// Dimmer timer expiration task.
// Dim the light by one step.
when (timer_expires(dimmer_timer))
{
    if (light_brightness > (MIN_BRIGHTNESS + bright_step)) {
        light_brightness -= bright_step;
        dimmer_timer = timer_period; // Set the Timer again
    } else {
        set_interior(FALSE);
        num_doors_open = 0;
    }
    io_out(io_int_light, light_brightness);
}

// Function to turn interior light on or off.
void set_interior(boolean light_on)
{
    if (light_on) {
        light_brightness = MAX_BRIGHTNESS;
        nv_light_on = ST_ON;
    } else {
        nv_light_brightness = MIN_BRIGHTNESS;
        nv_light_on = ST_OFF;
    }
}

```

Key Node

Node Definition

Output Network Variables

nv_key_state Ignition key state

Input I/O Objects

io_key_switch Ignition on/off switch (Bit input)

Functional Specification

- Initially the key is not in the ignition switch.
- When the ignition switch is turned on, the output network variable nv_key_state is set to ST_ON.
- When the ignition switch is turned off, the output network variable nv_key_state is set to ST_OFF.

key.nc

```
// Purpose: Example program. This application senses and
// reports the state of a car ignition key.
```

```
#include <io_types.h>
#include <snvt_lev.h>
```

```
// Network variable declarations
```

```
network output SNVT_lev_disc nv_key_state;
                // ST_ON if key is on
```

// I/O object declarations

```
const bit_t SWITCH_OFF_STATE = 1;
const bit_t SWITCH_ON_STATE = 0;
IO_7 input bit io_key_switch;
// Key switch
```

// Node reset task.

```
// Get the initial key state.
when (reset)
{
    nv_key_state = (io_in(io_key_switch)
                    == SWITCH_ON_STATE) ? ST_ON : ST_OFF;
    io_change_init(io_key_switch);
}
```

// Key turned on/off task

```
when (io_changes(io_key_switch))
{
    nv_key_state = (input_value
                    == SWITCH_ON_STATE) ? ST_ON : ST_OFF;
}
```

Headlight Switch Node

Node Definition

Output Network Variables

<code>nv_fade_mode</code>	Headlight fade mode
<code>nv_over_mode</code>	Headlight override mode
<code>nv_switch_state</code>	Headlight switch state

Input I/O Objects

<code>io_headlight_switch</code>	Manual on/off switch (Bit input)
<code>io_fade_mode_switch</code>	Left portion (fade mode) of 3-way mode indicator switch (Bit input)
<code>io_over_mode_switch</code>	Right portion (override mode) of 3-way mode indicator switch (Bit input)

Functional Specification

- When the on/off switch is moved to the on position, the output network variable `nv_switch_state` is set to `ST_ON`.
- When the on/off switch is moved to the off position, the output network variable `nv_switch_state` is set to `ST_OFF`.
- The headlight mode switch has three positions:
 - 1 Switch right : output network variable `nv_fade_mode` set to `ST_ON` (fade mode)
 - 2 Switch left: output network variable `nv_over_mode` set to `ST_ON` (override mode)

3 Switch center: output network variables

nv_fade_mode and nv_over_mode set to ST_OFF
(manual mode)

hswitch.nc

```
// Purpose: Example program. This application is the
// headlight switch for controlling a car headlight.

#include <io_types.h>
#include <snvt_lev.h>

// Network variable declarations
network output SNVT_lev_disc nv_fade_mode;
// ST_ON for fade mode. ST_OFF for
// manual mode
network output SNVT_lev_disc nv_over_mode;
// ST_ON for override mode. ST_OFF
// for manual mode
network output SNVT_lev_disc nv_switch_state = ST_OFF;
// ST_ON if headlight switch on

// I/O object declarations
#define SWITCH_OFF_STATE = 1;
#define SWITCH_ON_STATE = 0;
IO_7 input bit io_over_mode_switch;
// Right side of toggle switch
IO_8 input bit io_fade_mode_switch;
// Left side of toggle switch
IO_9 input bit io_headlight_switch;
// Headlight on/off switch
```

```

// Node reset task.
// Set the initial settings.
when (reset)
{
    // Get the mode switch position
    if (io_in(io_fade_mode_switch) == SWITCH_ON_STATE) {
        nv_fade_mode = ST_ON;
        nv_over_mode = ST_OFF;
    } else if (io_in(io_over_mode_switch) == SWITCH_ON_STATE) {
        nv_fade_mode = ST_OFF;
        nv_over_mode = ST_ON;
    } else {
        nv_fade_mode = ST_OFF;
        nv_over_mode = ST_OFF;
    }
    io_change_init(io_fade_mode_switch);
    io_change_init(io_over_mode_switch);
}

// Fade mode switch task
when (io_changes(io_fade_mode_switch))
{
    nv_fade_mode = (input_value ==
        SWITCH_ON_STATE) ? ST_ON : ST_OFF;
    nv_over_mode = ST_OFF;
}

// Override mode switch task
when (io_changes(io_over_mode_switch))
{
    nv_over_mode = (input_value ==
        SWITCH_ON_STATE) ? ST_OTHER : ST_OFF;
    nv_fade_mode = ST_OFF;
}

```

```
// Headlight on/off switch task
when (io_changes(io_headlight_switch) to
    SWITCH_ON_STATE)
{
    if (nv_switch_state == ST_ON)
        nv_switch_state = ST_OFF;
    else
        nv_switch_state = ST_ON;
}
```

Headlight Node

Node Definition

Input Network Variables

nv_key_state	Ignition key on/off indicator
nv_fade_mode	Headlight fade mode (on, off, override)
nv_over_mode	Headlight override mode (on, off)
nv_hl_state	Headlight on/off control

Output I/O Objects

io_headlight	Headlight brightness (Pulsewidth output)
--------------	--

Functional Specification

- Initially the headlight is off.
- When the ignition key is first turned on:
 - If the headlight switch is already in the on position, the headlight turns on at full brightness.
 - If the headlight switch is off, the light remains off.
- When the ignition key is already in the on position:
 - If the headlight switch is turned on, the headlight turns on at full brightness.
 - If the headlight switch is turned off, the headlight turns off. How it turns off is defined by the headlight mode as follows:
 - Manual mode: the light goes off immediately
 - Fade mode: the light slowly fades off
 - Override mode: the light goes off immediately
- When the ignition key is first turned off:
 - If the headlight is on, it turns off based on the mode as defined above except that if the headlight is in override mode it remains on.
- When the ignition key is already in the off position:
 - If the headlight switch is turned on, nothing happens.
 - If the headlight switch is turned off and the light is in override mode, the headlight immediately turns off.

headlight.nc

```
// Example program. This application is the headlight
// for a car.

#include <io_types.h>
#include <snvt_lev.h>

// Network variable declarations
network input SNVT_lev_disc nv_key_state = ST_OFF;
// ST_ON if ignition key on
network input SNVT_lev_disc nv_fade_mode = ST_OFF;
// ST_ON for fade mode
// ST_OFF for manual mode
network output SNVT_lev_disc nv_over_mode;
//ST_ON for override mode
//ST_OFF for manual mode
network input SNVT_lev_disc nv_hl_state = ST_OFF;
// ST_ON to cause headlight to turn on

// I/O object declarations
const pulsewidth_t MIN_BRIGHTNESS = 0;
const pulsewidth_t MAX_BRIGHTNESS = 255;
IO_0 output pulsewidth invert clock (7)
    io_headlight = 0;
// Headlight control

// Timer declarations
mtimer dimmer_timer; // Dimmer control sampling timer
```



```

// Global variable declarations
unsigned short int light_brightness = 0;
    // Brightness of the light
const short int bright_step = 5;
    // Brightness increment
const unsigned long int timer_period = 50;
    // Period for dimming lights

// Function prototype
void set_light(boolean light_on);

// Reset task
when (reset) {
    // Poll headlight network variable
    poll (nv_hl_state);
}

// Headlight on/off task
when (nv_update_occurs(nv_hl_state))
{
    if (nv_key_on == ST_ON)
        set_light(nv_hl_state == ST_ON);
    else if (nv_over_mode == ST_ON && nv_hl_state == ST_OFF)
        set_light(FALSE);
}

// Key change of state task
when (nv_update_occurs(nv_key_state))
{
    if (nv_key_state == ST_ON) {
        if (nv_hl_state == ST_ON)
            set_light(TRUE);
    } else if (nv_over_mode == ST_OFF)
        set_light(FALSE);
}

```

```

// Dimmer timer expiration task
when (timer_expires(dimmer_timer))
{
    // Knock the light down one notch
    if (light_brightness > bright_step) {
        light_brightness -= bright_step;
        dimmer_timer = timer_period;
        // Set the timer again
    } else
        light_brightness = MIN_BRIGHTNESS;

    io_out(io_int_light, light_brightness);
}

// Function to turn the headlight on or off
void set_light(boolean light_on)
{
    if (!light_on && (nv_fade_mode == ST_ON))
        // Turn light off with fade by
        // starting dimmer timer
        dimmer_timer = timer_period;
    else {
        dimmer_timer = 0;
        light_brightness =
            light_on ? MAX_BRIGHTNESS : MIN_BRIGHTNESS;
        io_out(io_headlight, light_brightness);
    }
}

```

Appendix B

Syntax Summary

This appendix provides a summary of NEURON C syntax.

Syntax Conventions

In this syntax section, syntactic categories (nonterminals) are indicated by *italic* type, and literal words and character set members (terminals) by **bold** type. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words “one of”.

NEURON C External Definitions

The language consists of basic blocks, called “external definitions”.

NEURON-C :

NEURON-C external-definition
external-definition

The external definitions are ANSI C objects, like data and function declarations, and NEURON C extensions, like I/O object declarations, and task declarations.

external-definition :

;
data-declaration
io-object-declaration ;
task-declaration
function-declaration

A data declaration is an ANSI C variable declaration.

data-declaration:

variable-declaration
variable-list ;

An I/O object declaration is similar to an ANSI C variable declaration. It may contain an initialization.

io-object-declaration :

modified-io-object-declarator variable-identifier = assign-expr
modified-io-object-declarator variable-identifier

The I/O object declaration begins with an I/O object declarator, possibly followed by one or more I/O object modifier clauses.

modifier-object-declarator :

io-object-declarator [*io-option-list*]

io-option-list :

io-option-list io-option
io-option

I/O Option Declarations

The I/O option declaration begins with a pin name, followed by the I/O object type.

io-object-declarator :

io-object-pin-name [*io-object-direction*] *io-object-type*

io-object-pin-name : IO_0 | IO_1 | IO_2 | IO_3 | IO_4 |
IO_5 | IO_6 | IO_7 | IO_8 | IO_9 | IO_10

io-object-direction : input | output

io-object-type :

bit
bitshift
byte
frequency
leveldetect
neurowire
nibble
oneshot
ontime
parallel
period
pulsecount
pulsewidth
quadrature
serial
totalcount
triac
triggeredcount

The I/O options (most only apply to a few specific object types).

io-option :

```
    baud ( assign-expr )
    clock ( assign-expr )
    clockedge ( clock-edge )
    ded
    invert
    kbaud ( assign-expr )
    long
    master
    mux
    numbits ( assign-expr )
    select ( io-object-pin-name )
    short
    slave
    slave_b
    sync ( io-object-pin-name )
    synchronized ( io-object-pin-name )
```

clock-edge : + | - | +- (+- is only valid on the NEURON 3120 CHIP)

Variable Declarations

The following is ANSI C variable declaration syntax.

variable-declaration-list :
 variable-declaration-list *variable-declaration*
 variable-declaration

variable-declaration :
 declaration-specifier-list *variable-list* ;
 declaration-specifier-list ;

variable-list :
 variable-list , *variable*
 variable

variable :
 declarator = *variable-initializer*
 declarator

variable-initializer :
 { *variable-initializer-list* , }
 { *variable-initializer-list* }
 assign-expr

variable-initializer-list :
 variable-initializer-list , *variable-initializer*
 variable-initializer

declaration-specifier-list :
 declaration-specifier-list *declaration-specifier*
 declaration-specifier

Declaration Specifiers

The ANSI C declaration specifiers are augmented in NEURON C by adding the connection information, the message tag specifier, network variable specifiers, and timer type specifiers.

declaration-specifier :

- connection-information*
- storage-class-specifier*
- cv-type-qualifier*
- msg_tag*
- net-var-type*
- timer-type*
- type-specifier*

type-specifier :

- enum-specifier*
- struct-or-union-specifier*
- type-identifier*
- type-keyword*

The *connection-information* feature is NEURON C specific. It allows the NEURON C programmer to communicate specific options directly to the network management tool for individual message tags and network variables.

connection-information :

- bind_info (bind-info-option-list)*
- bind_info ()*

bind-info-option-list :

- bind-info-option-list bind-info-option*
- bind-info-option*

bind-info-option :

auth (*configurable-keyword*)
authenticated (*configurable-keyword*)
auth
authenticated
bind
nonbind
offline
priority (*configurable-keyword*)
priority
nonpriority (*configurable-keyword*)
nonpriority
rate-est-keyword (*expression*)
service-type-keyword (*configurable-keyword*)
service-type-keyword

rate-est-keyword :

max_rate_est
rate_est

service-type-keyword :

ackd
unackd
unackd_rpt

configurable-keyword :

config
nonconfig

The ANSI C variable classes are augmented in NEURON C with the additional classes **config**, **eeprom**, **far**, **ram**, and **system**. The ANSI C **register** storage class is not supported.

class-keyword :

auto
config
eeprom
extern
far
ram
register (This keyword is not used in NEURON C)
static
system
typedef
volatile (This keyword is not used in NEURON C)

cv-type-qualifiers :

cv-type-qualifiers *cv-type-qualifier*
cv-type-qualifier

cv-type-qualifier :

const
volatile (This type-qualifier is not used in NEURON C)

The ANSI C data type keywords may appear in any order. Floating point types (**double** and **float**) are not supported in NEURON C.

type-keyword :

char
double (Floating point is not supported in NEURON C)
float (Floating point is not supported in NEURON C)
int
long
quad (This keyword is reserved for future implementations)
short
signed
unsigned
void

Network variables are declared with one of the following sequences of keywords. Network variables are specific to NEURON C.

net-var-type :

network	input	
network	input	sync
network	input	synchronized
network	output	
network	output	polled
network	output	sync
network	output	synchronized

Timer objects are declared with one of the following sequences of keywords. Timer objects are specific to NEURON C.

timer-type :

mtimer	[repeating]
stimer	[repeating]

Enumeration Syntax

The following is ANSI C enum type syntax.

enum-specifier :

enum *identifier* { *enum-value-list* }

enum { *enum-value-list* }

enum *identifier*

enum-value-list :

enum-const-list ,

enum-const-list

enum-const-list :

enum-const-list , *enum-const*

enum-const

enum-const :

variable-identifier = *assign-expr*

variable-identifier

Structure/Union Syntax

The following is ANSI C struct/union type syntax.

struct-or-union-specifier :

aggregate-keyword identifier { struct-decl-list }

aggregate-keyword { struct-decl-list }

aggregate-keyword identifier

aggregate-keyword :

struct

union

struct-decl-list :

struct-decl-list struct-declaration

struct-declaration

struct-declaration :

abstract-decl-specifier-list struct-declarator-list ;

struct-declarator-list :

struct-declarator-list , struct-declarator

struct-declarator

struct-declarator :

declarator

bitfield

bitfield :

declarator : assign-expr

: assign-expr

Declarator Syntax

The following is ANSI C declarator syntax. Pointers are not supported within network variables.

declarator :

* type-qualifier declarator

* *declarator*

sub-declarator

sub-declarator :

sub-declarator array-index-declaration

sub-declarator function-parameter-declaration

(*declarator*)

variable-identifier

array-index-declaration :

[*expression*]

[]

function-parameter-declaration :

formal-parameter-declaration

prototype-parameter-declaration

formal-parameter-declaration :

(*identifier-list*)

()

identifier-list :

identifier-list , *variable-identifier*

variable-identifier

prototype-parameter-declaration :

(*prototype-parameter-list* , ...)

(*prototype-parameter-list*)

prototype-parameter-list :

prototype-parameter-list , *prototype-parameter*

prototype-parameter

prototype-parameter :
 declaration-specifier-list *prototype-declarator*
 declaration-specifier-list

prototype-declarator :
 declarator
 abstract-declarator

Abstract Declarators

The following is ANSI C abstract declarator syntax.

abstract-declarator :
 * *cv-keywords* *abstract-declarator*
 * *abstract-declarator*
 * *cv-keywords*
 *
 abstract-sub-declarator

abstract-sub-declarator :
 (*abstract-declarator*)
 abstract-sub-declarator ()
 abstract-sub-declarator *prototype-parameter-declaration*
 abstract-sub-declarator *array-index-declaration* ()
 prototype-parameter-declaration
 array-index-declaration

abstract-type :
 abstract-decl-specifier-list *abstract-declarator*
 abstract-decl-specifier-list

abstract-decl-specifier-list :
 abstract-decl-specifier-list *abstract-decl-specifier*
 abstract-decl-specifier

abstract-decl-specifier :
 type-specifier
 cv-keyword

Function Declarators

The following is ANSI C function declarator syntax.

function-declaration :
 function-head compound-stmt

function-head :
 function-type-and-name parm-declaration-list
 function-type-and-name

function-type-and-name :
 declaration-specifier-list declarator declarator

parm-declaration-list :
 parm-declaration-list parm-declaration
 parm-declaration

parm-declaration :
 declaration-specifier-list parm-declarator-list ;

parm-declarator-list :
 parm-declarator-list , declarator
 declarator

Task Declarators

NEURON C also contains task declarations. A task declaration is a *when* clause list, followed by a task. A task is a compound statement (like an ANSI C function body).

task-declaration :
 when-clause-list *task*

when-clause-list :
 when-clause-list *when-clause*
 when-clause

when-clause :
 priority **when** *when-event*
 when *when-event*

task :
 compound-stmt

Conditional Events

In NEURON C, an event is an expression which may evaluate to either TRUE or FALSE. This extends the ANSI C concept of conditional expressions with special “built-in functions” to test special NEURON CHIP firmware events.

when-event :
 (**reset**)
 parenthesized-expr

predefined-event :
 (**flush_completes**)
 (**offline**)
 (**online**)
 (**wink**)
 (*complex-event*)

complex-event :

io-event
message-event
net-var-event
timer-event

io-event :

io_update_occurs (*variable-identifier*)
io_change_event
io_changes (*variable-identifier*)

message-event :

message-event-keyword (*expression*)
message-event-keyword

message-event-keyword :

msg_arrives
msg_completes
msg_fails
msg_succeeds
resp_arrives

net-var-event :

nv-event-keyword (*variable-identifier* [*expr*])
nv-event-keyword (*variable-identifier*)
nv-event-keyword

nv-event-keyword :

nv_update_completes
nv_update_fails
nv_update_occurs
nv_update_succeeds

timer-event :

timer_expires (*variable-identifier*)
timer_expires

Statements

The following is ANSI C statement syntax.

compound-stmt :

```
{ variable-declaration-list statement-list }  
{ variable-declaration-list }  
{ statement-list }  
{ }
```

statement-list :

```
statement-list statement  
statement
```

statement :

```
complete-stmt  
incomplete-stmt
```

complete-stmt :

```
compound-stmt  
label : complete-stmt  
break ;  
continue ;  
do statement while-clause ;  
for-head complete-stmt  
goto identifier ;  
if-else-head complete-stmt  
switch complete-stmt  
return ;  
return expression ;  
while-clause complete-stmt  
expression ;  
;
```

incomplete-stmt :

```
label : incomplete-stmt  
for-head incomplete-stmt  
if-else-head incomplete-stmt  
if-head statement  
switch incomplete-stmt  
while-clause incomplete-stmt
```

label :
 case *expression*
 default
 identifier

if-else-head :
 if-head **complete-stmt** **else**

if-head :
 if *parenthesized-expr*

for-head :
 for ([*expression*] ; [*expression*] ; [*expression*])

switch :
 switch *parenthesized-expr*

while-clause :
 while *parenthesized-expr*

ANSI C Expressions

The following is ANSI C expression syntax.

parenthesized-expr :
 (*expression*)

expression :
 expression , *assign-expr*
 assign-expr

assign-expr :
 choice-expr *assign-op* *assign-expr*
 choice-expr

assign-op : = | |= | ^= | &= | <<= | >>=

 | /= | *= | %= | += | -=

choice-expr :
 logical-or-expr ? *expression* : *choice-expr*
 logical-or-expr

logical-or-expr :
 logical-or-expr || *logical-and-expr*
 logical-and-expr

logical-and-expr :
 logical-and-expr && *bit-or-expr*
 bit-or-expr

bit-or-expr :
 bit-or-expr | *bit-xor-expr*
 bit-xor-expr

bit-xor-expr :
 bit-xor-expr ^ *bit-and-expr*
 bit-and-expr

bit-and-expr :
 bit-and-expr & *equality-comparison*
 equality-comparison

equality-comparison :
 equality-comparison == *relational-comparison*
 equality-comparison != *relational-comparison*
 relational-comparison

relational-comparison :
 relational-comparison *relational-op* *io-change-by-to-expr*
 io-change-by-to-expr

relational-op : < | <= | >= | >

io-change-by-to-expr :
 io-change-event **by** *shift-expr*
 io-change-event **to** *shift-expr*
 shift-expr

shift-expr :
 shift-expr shift-op additive-expr
 additive-expr

shift-op : << | >>

additive-expr :
 additive-expr add-op multiplicative-expr
 multiplicative-expr

add-op : + | -

multiplicative-expr :
 multiplicative-expr mul-op cast-expr
 cast-expr

mul-op : * | / | %

cast-expr :
 (*abstract-type*) *cast-expr*
 unary-expr

unary-expr :
 unary-op cast-expr
 sizeof unary-expr
 sizeof (abstract-type)
 predefined-event

unary-op : * | & | ! | ~ | + | - | ++ | --

postfix-expr :
 postfix-expr [expression]
 postfix-expr -> identifier
 postfix-expr . identifier
 postfix-expr ++
 postfix-expr --
 postfix-expr actual-parameters
 primary-expr

```

actual-parameters :
    ( actual-parameter-list )
    ( )

```

```

actual-parameter-list :
    actual-parameter-list , assign-expr
    assign-expr

```

Built-in Variables and Functions

In addition to the ANSI C definitions of a primary expression, NEURON C adds some built-in variables and built-in functions. Note also that NEURON C removes *float-constant* from the standard list of primary expressions.

```

primary-expr :
    parenthesized-expr
    integer-constant
    concatenated-string-constant
    variable-identifier
    builtin-variables
    builtin-functions
    actual-parameters
    msg-call-kwd ( )

```

stet → *builtin-functions* *actual-parameters*

```

concatenated-string-constant :
    concatenated-string-constant
    string-constant
    string-constant

```

```

builtin-variables :
    input_is_new
    input_value
    msg-name-kwd . variable-identifier

```

nv_array_index
nv_in_addr

```

msg-name-kwd :
    msg_in | msg_out | resp_in | resp_out

```


builtin-functions :

```
abs
bcd2bin
bin2bcd
io_change_init
io_in
io_in_ready
io_out
io_out_ready
io_out_request
io_select
io_set_clock
is_bound
max
memcpy
memset
min
poll
sleep
    io_in_request
    io_preserve_input
```

msg-call-kwd :

```
msg_alloc | msg_alloc_priority |
msg_cancel | msg_free | msg_receive |
msg_send | resp_alloc | resp_cancel |
resp_free | resp_receive | resp_send
```

Limits.h

The contents of the standard include file <limits.h> are given below.

```
#define CHAR_BIT      8
#define CHAR_MAX      127
#define CHAR_MIN      (-128)

#define LONG_MAX      32767
#define LONG_MIN      (-32768)

#define MB_LEN_MAX    2

#define SCHAR_MAX      127
#define SCHAR_MIN      (-128)
#define UCHAR_MAX      255

#define SHRT_MAX      127
#define SHRT_MIN      (-128)

#define USHRT_MAX      255

#define INT_MAX      127
#define INT_MIN      (-128)

#define UINT_MAX      255

#define ULONG_MAX      65535
```

Appendix C

NEURON C Language

Reference

This appendix provides the following reference sections on NEURON C:

<i>Section</i>		<i>Page No.</i>
C.1	Predefined Events	C-2
C.2	Functions	C-22
C.3	Network Variable Declarations	C-76
C.4	Timer Declarations	C-82
C.5	Built-in Variables and Objects	C-83
C.6	Reserved Words	C-90
C.7	I/O Objects	C-95

C.1 Predefined Events

Predefined events are represented by unique keywords, listed in the table below. Some predefined events, such as the I/O events, may be followed by a modifier that narrows the scope of the event. If the modifier is optional and not supplied, any event of that type qualifies. The following table lists events by functional group.

System / Scheduler	Network Variables
offline	nv_update_completes
online	nv_update_fails
reset	nv_update_occurs
timer_expires	nv_update_succeeds
wink	
	Messages
Input/Output	msg_arrives
io_changes	msg_completes
io_update_occurs	msg_fails
io_in_ready	msg_succeeds
io_out_ready	resp_arrives
Sleep	
flush_completes	

Within a single program, the following predefined events, which reflect state transitions of the application processor, can appear in no more than one when clause:

- reset
- offline
- online
- timer_expires (unqualified)
- wink

All other predefined events can be used in multiple when clauses. Predefined events (except for the reset event) can also be used in any NEURON C expression.

Event Directory

The following pages list NEURON C events alphabetically, providing relevant syntax information and a detailed description of each event.

flush_completes

EVENT

flush_completes

The **flush_completes** event evaluates to TRUE when all outgoing transactions have been completed and no more incoming messages remain to be processed. For unacknowledged messages, “completed” means that the message has been transmitted by the Media Access Control (MAC) layer. For acknowledged messages, “completed” means that the completion code has been processed. In addition, all network variable updates have completed.

See also the discussion of *sleep mode* in Chapter 5.

EXAMPLE:

```
...
flush();
...
when (flush_completes) {
    sleep();
}
```

io_changes

EVENT

io_changes (*io_object_name*) [**to** *expr* | **by** *expr*]

The **io_changes** event evaluates to TRUE when the value read from the I/O object specified by *io_object_name* changes state. The change can be one of three types:

- a change *to* a specified value
- a change *by* (at least) a specified amount (absolute value)
- any change (an unqualified change)

The *reference value* is the value read the last time the change event evaluated to TRUE. For the unqualified `io_changes` event, a state change occurs when the current value is different from the reference value.

A task can access the input value for the I/O object through the `input_value` keyword. `input_value` is always a long.

For bit, byte and nibble I/O objects, changes are not latched. The change must persist until the `io_changes` event is processed. The `leveldetect` input object can be used to latch changes that may not persist until the `io_changes` event can be processed.

Following are more detailed descriptions of the elements of the above syntax:

io_object_name is the I/O object name (see *Section C.7*). I/O objects of the following input object types can be used in an unqualified change event. The *by* and *to* options can also be used where noted.

```
bit (to)
byte (by, to)
leveldetect (to)
nibble (by, to)
ontime (by)
period (by, to)
pulsecount (by)
quadrature (by)
dualslope (by)
```

to expr where *expr* is a NEURON C expression. The *to* option specifies the value of the I/O state necessary for the `io_changes` event to become TRUE. (The compiler accepts an unsigned long value for the expression. However, each I/O object type has its own range of meaningful values.)

by expr where *expr* is a *Neuron C* expression. The *by* option compares the current value with the reference value. The `io_changes` event becomes TRUE when the difference (absolute value) between the current value and the reference value is greater than or equal to *expr*.

The default initial reference value used for comparison purposes is zero. You can set the initial value by calling the `io_change_init()` function. If an explicit reference value is passed to `io_change_init()`, that value is used as the initial reference value: `io_change_init(io_object_name, value)`. If no explicit value is passed to `io_change_init()`, the I/O object's current value is used as the initial value: `io_change_init(io_object_name)`.

EXAMPLE 1:

```
IO_0 input bit push_button;

when (io_changes(push_button) to 0)
{
  ...
}
```

EXAMPLE 2:

```
IO_7 input pulsecount total_ticks;

when (io_changes(total_ticks) by 100)
{
  ...
}
```

io_in_ready

EVENT

io_in_ready (*parallel_io_object_name*)

parallel_io_object_name is the parallel I/O object name (see *Section C.7*).

The **io_in_ready** event evaluates to TRUE when a block of data is available to be read on the parallel bus. The application then calls `io_in()` to retrieve the data. (See also the discussion of *parallel I/O* in Chapter 2 and the *parallel I/O object* in *Section C.7*.)

EXAMPLE:

```
when (io_in_ready(io_bus))
{
    io_in(io_bus, &piofc);
}
```

io_out_ready

EVENT

`io_out_ready (parallel_io_object_name)`

parallel_io_object_name is the parallel I/O object name (see *Section C.7*).

The `io_out_ready` event evaluates to TRUE whenever the parallel bus is in a state where it can be written to and the `io_out_request()` function has been previously invoked (See also the discussion of *parallel I/O* in Chapter 2 and the *parallel I/O object* in *Section C.7*.)

EXAMPLE:

```
when (...)  
{  
    io_out_request(io_bus);  
}  
  
when (io_out_ready(io_bus))  
{  
    io_out(io_bus, &piofc);  
}
```

io_update_occurs

EVENT

`io_update_occurs (io_object_name)`

io_object_name is the I/O object name (see *Section C.7*).

The `io_update_occurs` event evaluates to TRUE when the input object specified by *io_object_name* has an updated value. The `io_update_occurs` event applies only to timer/counter input object types (ontime, period, pulsecount, quadrature, and totalcount), as follows:

<i>I/O Object</i>	<i>io_update_occurs evaluates to TRUE after:</i>
ontime	the edge is detected defining the end of a period
period	the edge is detected defining the end of a period
pulsecount	every 0.8388608 seconds
quadrature	the encoder position changes

dualslope the A/D conversion is complete

An input object may have an *updated* value that is actually the *same* as its previous value. To detect *changes* in value, use the `io_changes` event. A given I/O object cannot be included in when clauses with both `io_update_occurs` and `io_changes` events.

A task can access the updated value for the I/O object through the `input_value` keyword. The value `input_value` is always a long.

EXAMPLE:

```
IO_7 input ontime io_thermistor;

when (io_update_occurs(io_thermistor))
{
    ...
}
```

msg_arrives

EVENT

msg_arrives [(*message_code*)]

message_code is an optional integer message code. If this field is omitted, the event is TRUE for receipt of any message.

The **msg_arrives** event evaluates to TRUE when a message arrives. This event can be qualified by a specific message code specified by the sender of the message. See Chapter 4 for a list of message code ranges.

EXAMPLE:

```
when (msg_arrives(10))
{
    ...
}
```

msg_completes

EVENT

msg_completes [(*message_tag*)]

message_tag is an optional message tag. If this field is omitted, the event is TRUE for any message.

The **msg_completes** event evaluates to TRUE when an outgoing message completes (that is, either succeeds or fails). This event can be qualified by a specific message tag.

Checking the completion event (**msg_completes**, **msg_fails**, **msg_succeeds**) is optional by message tag.

If a program checks for either the `msg_succeeds` or `msg_fails` event, it must check for *both* events. The alternative is to check only for `msg_completes`.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_completes(tag_out))
{
    ...
}
```

msg_fails

EVENT

msg_fails [(*message_tag*)]

message_tag is an optional message tag. If this field is omitted, the event is TRUE for any message.

The **msg_fails** event evaluates to TRUE when a message fails to be acknowledged after all retries have been attempted. This event can be qualified by a specific message tag.

Checking the completion event (`msg_completes` or `msg_fails/msg_succeeds`) is optional by message tag. If a program checks for either the `msg_succeeds` or `msg_fails` event, it must check for *both* events. The alternative is to check only for `msg_completes`.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_fails(tag_out))
{
    ...
}
```

msg_succeeds [(*message_tag*)]

message_tag is an optional message tag. If this field is omitted, the event is TRUE for any message.

The **msg_succeeds** event evaluates to TRUE when a message is successfully sent (see Table 4-2 for the definition of success). This event can be qualified by a specific message tag.

Checking the completion event (**msg_completes**, **msg_fails**, **msg_succeeds**) is optional by message tag. If a program checks for either the **msg_succeeds** or **msg_fails** event, it must check for *both* events. The alternative is to check only for **msg_completes**.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_send();
...
when (msg_succeeds(tag_out))
{
    ...
}
```

nv_update_completes

EVENT

nv_update_completes [(*network_var*)]

network_var is a network variable identifier, network variable array name, or network variable array element. If the parameter is omitted, the event is TRUE when any network variable update completes.

The **nv_update_completes** event evaluates to TRUE when an output network variable update completes (that is, either fails or succeeds) or a poll operation completes. Checking the completion event (**nv_update_completes** or **nv_update_fails** / **nv_update_succeeds**) is optional by network variable.

If an array name is used, then each element of the array will be checked for completion. The event will occur once for each element as applicable. An individual element may be checked with use of an array index. When **nv_update_completes** is TRUE, the built-in variable 'nv_array_index' (type short int) may be examined to obtain the element's index to which the event applies.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check *only* for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

EXAMPLE:

```
network output int humidity;
...
humidity = 32;
...
when (nv_update_completes(humidity))
{
    ...
}
```

nv_update_fails

EVENT

nv_update_fails [(*network_var*)]

network_var is a network variable identifier, array name, or array element. If the parameter is omitted, the event is TRUE when any network variable update fails.

The **nv_update_fails** event evaluates to TRUE when an output network variable update or poll fails (see *Table 4-2* for the definition of success).

If an array name is used, then each element of the array will be checked for failure. The event will occur once for each element as applicable. An individual element may be checked with use of an array index. When **nv_update_fails** is TRUE, the built-in variable 'nv_array_index' (type short int) may be examined to obtain the element's index to which the event applies.

Checking the completion event (**nv_update_completes** or **nv_update_fails/nv_update_succeeds**) is optional by network variable.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check only for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

EXAMPLE:

```
network output int humidity;
...
humidity = 32;
...
when (nv_update_fails (humidity))
{
    ...
}
```

nv_update_occurs

EVENT

nv_update_occurs [(*network_var*)]

network_var is a network variable identifier, array name, or array element. If the parameter is omitted, the event is TRUE for any network variable update.

The **nv_update_occurs** event evaluates to TRUE when a value has been received for an input network variable.

If an array name is used, then each element of the array will be checked to see if a value has been received. The event will occur once for each element as applicable. An individual element may be checked with use of an array index. When **nv_update_occurs** is TRUE, the built-in variable 'nv_array_index' (type short int) may be examined to obtain the element's index to which the event applies.

EXAMPLE:

```
network input boolean switch_state;

when (nv_update_occurs (switch_state))
{
    ...
}
```

nv_update_succeeds

EVENT

nv_update_succeeds [(*network_var*)]

network_var is a network variable identifier, array name, or array element. If the parameter is omitted, the event is TRUE when any network variable update succeeds.

The **nv_update_succeeds** event evaluates to TRUE when an output network variable update has been successfully sent or a poll succeeds.

If an array name is used, then each element of the array will be checked for success. The event will occur once for each element as applicable. An individual element may be checked with use of an array index. When **nv_update_succeeds** is TRUE, the built-in variable 'nv_array_index' (type short int) may be examined to obtain the element's index to which the event applies.

Checking the completion event (**nv_update_completes** or **nv_update_fails** / **nv_update_succeeds**) is optional by network variable.

If a program checks for the **nv_update_succeeds** event, it must check for the **nv_update_fails** event as well. The alternative is to check only for **nv_update_completes**. A program is also permitted to check only for **nv_update_fails** as long as there is no use of **nv_update_completes** or **nv_update_succeeds** for *any* network variable.

EXAMPLE:

```
network output int humidity;
...
humidity = 32;
...
when (nv_update_succeeds (humidity))
{
    ...
}
```


offline

The **offline** event evaluates to TRUE only if the node is online and an offline network management message is received, or when a program calls `go_offline`. The **offline** event is handled as the first priority when clause. It can be used in no more than one when clause in a program.

The **offline** event can be used to place a network node offline in case of an emergency, for maintenance, or in response to some other system-wide condition. After execution of this event and its task, the application program halts until the node is reset or brought back online. Once offline, a node responds only to the reset or online commands from a network management tool. Network variables on an offline node cannot be polled.

If this event is checked for outside of a when clause, the programmer can confirm to the scheduler that the application program is going offline by calling the `offline_confirm()` function (see the *Going offline in Bypass Mode* section in Chapter 5).

When an application goes offline, all outstanding transactions are terminated. To ensure that any outstanding transactions complete normally, the application can call `flush_wait()` in the `when (offline)` task.

EXAMPLE:

```
when (offline)
{
    flush_wait();
    // process shut-down command
}
when (online)
{
    // start-up again, poll inputs
}
```

online

EVENT

online

The **online** event evaluates to TRUE only if the node is offline and an online network management message is received. The **online** event can be used in no more than one when clause in a program. The task associated with the **online** event in a when clause can be used to bring a node back into operation in a well-defined state.

EXAMPLE:

```
when (offline)
{
    flush_wait();
    // process shut-down command
}
when (online)
{
    // resume operation
}
```

reset

EVENT

reset

The **reset** event evaluates to TRUE the first time this event is evaluated after the NEURON CHIP is reset. (I/O object and global variable initializations are performed before processing any events.) The **reset** event task is always executed first after reset of the NEURON CHIP. The **reset** event can be used in no more than one when clause in a program.

The code in a reset task is limited in size. If you need more code than the compiler permits, move some or all of the code within the reset task to a function called from the reset task.

The `power_up()` function can be called in a reset clause to determine whether the reset was due to power-up, or to some other cause such as a hardware reset, software reset or watchdog timer reset.

EXAMPLE:

```
when (reset)
{
    // poll state of all inputs
}
```

resp_arrives

EVENT

resp_arrives [(*message_tag*)]

message_tag is an optional message tag. If this field is omitted, the event is TRUE for receipt of any response message.

The `resp_arrives` event evaluates to TRUE when a response arrives. This event can be qualified by a specific message tag.

EXAMPLE:

```
msg_tag tag_out;
...
msg_out.tag = tag_out;
msg_out.service = REQUEST;
msg_send();
...
when (resp_arrives(tag_out))
{
    ...
}
```

timer_expires

EVENT

timer_expires [(*timer_name*)]

timer_name is an optional timer object. If this field is omitted, the event is TRUE as long as any timer object has expired.

The **timer_expires** event evaluates to TRUE when a previously declared timer object expires. If the *timer_name* option is not included, the event is an “unqualified” **timer_expires** event. Unlike all other predefined events, which are TRUE only once, the unqualified **timer_expires** event remains TRUE as long as any timer object has expired. This event can be cleared only by checking for specific timer expiration events.

EXAMPLE:

```
mtimer countdown;
...
countdown = 100;
...
when (timer_expires(countdown))
{
    ...
}
```

wink

EVENT

wink

The **wink** event evaluates to TRUE whenever a network management **wink** command is received from a network management tool. The node can be configured or unconfigured, but it must have a program running on it.

The **wink** event is unique in that it can evaluate to TRUE even though the node is unconfigured. This event facilitates installation by allowing an unconfigured node to perform an action in response to the network management tool's **wink** command.

EXAMPLE:

```
when (wink)
{
...io_out(io_indicator_light, ON);
}
```

C.2 Functions

The following pages list NEURON C functions, providing syntax information, descriptions, and examples of each function. Some functions are “built-in” functions. This means they are used as if they were function calls, but they have special behavior depending on their context. The remainder are library calls. Some library calls have function prototypes in one of the standard include files, as noted. These are:

- ACCESS.H
- ADDRDEFS.H
- CONTROL.H
- STATUS.H
- STDLIB.H

The remainder of the functions and built-in functions derive their prototypes from ECHELON.H, an include file that is automatically incorporated in each compilation. Except for ECHELON.H, you must incorporate the necessary include file to use the function. Although some of the following function descriptions list both an include file and a prototype, you should only specify the `#include` directive. The prototype is shown only for reference.

Note that any existing application program developed for a NEURON 3120 CHIP that uses any of the functions which are brought in from a system library will require more EEPROM memory on a NEURON 3120 CHIP than on a NEURON 3150 CHIP. This is because these functions have been moved from the ROM portion of the NEURON CHIP firmware to a system library. Examination of the link map provides a measure of the EEPROM memory used by these functions. See the *System Library on a NEURON 3120 CHIP* section in Chapter 6 for more detailed information on how to create and examine a link map to obtain a measure of the NEURON 3120 CHIP EEPROM usage required for these functions. Also see the *LONBUILDER User's Guide* for additional information on the link map.

The following functions are automatically called by the compiler:

_bitf_sign_ext() (any use of a *signed* bitfield)
 _memcpy16() (memcpy() or struct assign length >= 256)
 _memset16() (memset() with length >= 256)
 _msg_data_blockget() (memcpy() out of msg_in.data)
 _msg_in_addr_ptr() (any use of msg_in.addr)
 _msg_out_addr_ptr() (any use of msg_out.dest_addr)
 _neurowire_slave() (any io_in() or io_out() for a neurowire slave object)
 _resp_data_blockset() (memcpy() into resp_out.data)

The table on the following page lists functional groups in NEURON C and specifies which are:

- built into the compiler
- contained in the system image
- placed in a system library on the NEURON 3120 CHIP

For easy reference, the individual descriptions of each function are combined into a single alphabetical listing on the pages which follow the table.

The following functions are automatically called by the compiler:

_bitf_sign_ext() (any use of a *signed* bitfield)
 _dualslope_input() (any io_in() for a dualslope object)
 _dualslope_start() (any io_in_request() for a dualslope object)
 _edgelog_input() (any io_in() for an edgelog object)
 _io_set_clock_x2() (any io_set_clock() for an edgelog object)
 _ir_input() (any io_in() for an infrared object)
 _magcard_input() (any io_in() for a magcard object)
 _memcpy16() (memcpy() or struct assign length >= 256)
 _memset16() (memset() with length >= 256)
 _msg_data_blockget() (memcpy() out of msg_in.data)
 _msg_in_addr_ptr() (any use of msg_in.addr)
 _msg_out_addr_ptr() (any use of msg_out.dest_addr)
 _muxbus_read() (any io_in() for a muxbus object)
 _muxbus_reread() (any io_in() for a muxbus object)
 _muxbus_rewrite() (any io_out() for a muxbus object)
 _muxbus_write() (any io_out() for a muxbus object)
 _neurowire_slave() (any io_in() or io_out() for a neurowire slave object)
 _resp_data_blockset() (memcpy() into resp_out.data)
 _resp_in_addr_ptr() (any use of resp_in.addr)

■ Page C-24 Replace

Utilities	Network Variables/ Messages	Installation	Operating System / Scheduler
abs() ¹	is_bound() ¹	access_address() ³	delay() ²
bed2bin() ³	msg_alloc() ¹	access_domain() ³	flush_wait() ³
bin2bed() ³	msg_alloc_priority() ¹	access_nv() ³	go_offline() ²
max() ¹	msg_cancel() ¹	addr_table_index() ¹	go_unconfigured() ³
memcpy() ¹	msg_free() ¹	nv_table_index() ¹	offline_confirm() ²
memset() ¹	msg_receive() ¹	update_address() ³	post_events() ²
min() ¹	msg_send() ¹	update_clone_domain() ³	power_up() ³
muldiv() ³	poll() ¹	update_config_data() ³	scaled_delay() ²
muldivs() ³	resp_alloc() ¹	update_domain() ³	timers_off() ²
random() ²	resp_cancel() ¹	update_nv() ³	watchdog_update() ²
refresh_memory() ²	resp_free() ¹	Input/Output	Error Handling
reverse() ³	resp_receive() ¹	io_change_init() ¹	application_restart() ²
Sleep Mode	resp_send() ¹	io_edgelog_preload() ³	clear_status() ²
flush() ²		io_in() ¹	error_log() ²
flush_cancel() ²		io_in_ready() ¹	node_reset() ²
sleep() ¹		io_in_request() ³	retrieve_status() ³
		io_out() ¹	
		io_out_request() ¹	
		io_preserve_input() ³	
		io_select() ¹	
		io_set_clock() ¹	
		io_set_direction() ¹	
¹ - a built-in function ² - the function is in the system image on both the NEURON 3150 and NEURON 3120 CHIP ³ - the function is in the system image on the NEURON 3150 CHIP and in a system library on the NEURON 3120 CHIP			

Function Directory

abs()

Built-in Function

type abs (a);

The **abs()** built-in function returns the absolute value of *a*. The argument *a* can be of type short or long. The return type is unsigned short if *a* is short, or unsigned long if *a* is long.

EXAMPLE :

```
int i;
long l;

i = abs(-3);
l = abs(-300);
```

access_address()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
const address_struct * access_address(int index);
```

The **access_address()** function returns a `const` pointer to the address structure which corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to C pointers, except that the pointer cannot be used for writes.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE :

```
#include <addrdefs.h>
#include <access.h>
address_struct addr_copy;

addr_copy = * (access_address(2));
```

access_domain()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
const domain_struct * access_domain(int index);
```

The **access_domain()** function returns a **const** pointer to the domain structure which corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to C pointers, except that the pointer cannot be used for writes.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE :

```
#include <addrdefs.h>
#include <access.h>
domain_struct domain_copy;

domain_copy = *
    (access_domain(0));
```

access_nv()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
const nv_struct * access_nv(int index);
```

The **access_nv()** function returns a **const** pointer to the network variable configuration structure which corresponds to the index parameter. This pointer can be stored, used to perform a structure copy, or used in other ways common to NEURON C pointers, except that the pointer cannot be used for writes.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE :

```
#include <addrdefs.h>
#include <access.h>
network output int my_nv;
nv_struct nv_copy;

nv_copy = *(access_nv(nv_table_index(my_nv)));
```

addr_table_index()

Built-in Function

The **addr_table_index()** built-in function is used to determine the address table index of a message tag as allocated by the NEURON C compiler. The returned value is in the range of 0 to 14.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks* Engineering Bulletin (part no. 005-0022-01) for further information.

EXAMPLE :

```
int mt_index;
msg_tag my_mt;

mt_index = addr_table_index(my_mt);
```

application_restart()

FUNCTION

```
#include <control.h>
void application_restart (void);
```

The **application_restart()** function re-starts the application program running on the application processor only. The network and MAC processors are unaffected. When an application is restarted, the **when(reset)** event becomes **TRUE**.

EXAMPLE :

```
#define MAX_ERRS 50 int error_count;
...
when (error_count > MAX_ERRS)
{
    application_restart();
}
```

bcd2bin()

Built-in Function

`unsigned long bcd2bin (struct bcd * a);`

```
struct bcd {  
    unsigned d1:4,  
             d2:4,  
             d3:4,  
             d4:4,  
             d5:4,  
             d6:4;  
};
```

The **bcd2bin()** built-in function converts a binary coded decimal structure to a binary number. The structure definition is built into the compiler. The most significant digit is *d1*. Note that *d1* should always be 0.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
struct bcd digits;  
unsigned long value;  
memset(&digits, 0, 3);  
digits.d3=1;  
digits.d4=2;  
digits.d5=3;  
digits.d6=4;  
value = bcd2bin(&digits);  
//value now contains 1234
```

bin2bcd()

Built-in Function

void bin2bcd (unsigned long value, struct bcd * p);

struct bcd (see *bcd2bin*, above)

The **bin2bcd()** built-in function converts a binary number to a binary coded decimal structure.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
struct bcd digits;
unsigned long value;
...
value = 1234;
bin2bcd(value, &digits);
//digits.d1 now contains 0
//digits.d2 now contains 0
//digits.d3 now contains 1
//digits.d4 now contains 2
//digits.d5 now contains 3
//digits.d6 now contains 4
```

clear_status()

FUNCTION

```
#include <status.h>
void clear_status (void);
```

The **clear_status()** function clears a subset of the information in the status structure (see the **retrieve_status()** function described later in this appendix). The information cleared is the statistics information, the reset cause register, and the error log.

EXAMPLE:

```
when (timer_expires(statistics_reporting_timer))
{
    retrieve_status(status_ptr);
    // get current statistics
    report_statistics(status_ptr);
    // check it all out
    clear_status();
}
```

delay()

Function

void **delay** (unsigned long count);

count is a value between 1 and 33333. The formula for determining the duration of the delay is based on count and the input clock (see below).

The **delay()** function allows an application to suspend processing for a given time. This function provides more precise timing than can be achieved with application timers.

The formulas for determining the duration of the delay are:

<i>Input Clock</i>	<i>Delay in microseconds</i>
10 MHz	$0.6 * (\max(1, \text{count}) * 42 + 106)$
5 MHz	$1.2 * ((\max(1, \text{floor}(\text{count}/2)) * 42) + 144)$
2.5 MHz	$2.4 * ((\max(1, \text{floor}(\text{count}/4)) * 42) + 164)$
1.25 MHz	$4.8 * ((\max(1, \text{floor}(\text{count}/8)) * 42) + 184)$
625kHz	$9.6 * ((\max(1, \text{floor}(\text{count}/16)) * 42) + 204)$

This formula yields durations in the range of 88.8 microseconds to 840 milliseconds by increments of 25.2 microseconds with a 10 MHz input clock. Using a count above 33,333 may cause the watchdog timer to time out. (See also the `scaled_delay()` function, which generates a delay that scales with the input clock.)

EXAMPLE :

```

IO_4 input bit io_push_button;
boolean debounced_button_state;

when(io_changes(io_push_button))
{
    delay(400);
    //delay approx. 10 msec at any clock rate
    debounced_button_state=(boolean)io_in(io_push_button);
}

```

error_log()

FUNCTION

```

#include <control.h>
void error_log (unsigned int error_num);

```

error_num is a decimal number between 1 and 127.

The **error_log()** function writes the error number into a dedicated location in EEPROM. Network management tools can use the query status network diagnostic command to read the last error. The NEURON C Debugger maintains a log of the last 25 error messages. On a NEURON Emulator, the NEURON CHIP firmware adds a delay of up to 70 msec between writes to the error log to give the PC time to retrieve the last value.

Appendix F lists the error numbers that are used by the NEURON CHIP firmware. These are in the range 128 ... 255. The application uses error numbers 1 ... 127.

EXAMPLE :

```

#define MY_ERROR_CODE 1
...
when (nv_update_fails)
{
    error_log(MY_ERROR_CODE);
}

```

flush()

Function

```
#include <control.h>
void flush (boolean comm_ignore);
```

comm_ignore indicates whether the NEURON CHIP should ignore communications channel activity. Specify TRUE if the NEURON CHIP should ignore any further incoming messages. Specify FALSE if the NEURON CHIP should continue to accept incoming messages.

The **flush()** function causes the NEURON CHIP to monitor the status of all outgoing and incoming messages. The **flush_completes** event becomes TRUE when all outgoing transactions have been completed and no more incoming messages are outstanding. For unacknowledged messages, “completed” means that the message has been fully transmitted by the MAC layer. For acknowledged messages, “completed” means that the completion code has been processed. In addition, all network variable updates must be propagated before the flush can be considered complete.

EXAMPLE:

```
boolean nothing_to_do;
...
when (nothing_to_do)
{
    // Getting ready to sleep
    flush(TRUE);
}

when (flush_completes)
{
    // Go to sleep
    sleep();
}
```

flush_cancel()

FUNCTION

```
#include <control.h>
void flush_cancel (void);
```

The **flush_cancel()** function cancels a flush in progress.

EXAMPLE:

```
boolean nothing_to_do;
...
when (nv_update_occurs)
{
    if (nothing_to_do) {
        // was getting ready to sleep but received an input NV
        nothing_to_do = FALSE;
        flush_cancel();
    }
}
```

flush_wait()

FUNCTION

```
#include <control.h>
void flush_wait (void);
```

The **flush_wait()** function causes an application program to enter preemption mode, during which all outstanding network variable and message transactions are completed. When a program switches from asynchronous to direct event processing, **flush_wait()** is used to ensure that all pending asynchronous transactions are completed before direct event processing begins.

During preemption mode, only pending completion events (for example, **msg_completes**, **nv_update_fails**) and pending response events (for example, **resp_arrives**, **nv_update_occurs**) are processed. When this processing is complete, **flush_wait()** returns. The application program can now process network variables and messages directly and need not concern itself with outstanding completion events and responses from earlier transactions.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
msg_tag TAG1;
network output int NV1;

when (x)
{
    msg_out.tag = TAG1;
    msg_out.code = 3;
    msg_send();

    flush_wait();

    NV1 = 3;
    while (TRUE) {
        post_events();
        if (nv_update_completes(NV1))
            break;
    }

    when (msg_completes (TAG1))
    {
        ...
    }
}
```

go_offline()

FUNCTION

```
#include <control.h>
void go_offline (void);
```

The **go_offline()** function takes an application offline. This function call has the same effect on the node as receiving a network management offline request. The offline request takes effect as soon as the task that called **go_offline()** is exited. When that task is exited, the **when (offline)** task is executed and the application stops.

When a network management online request is received, the **when (online)** task is executed and the application resumes execution.

When an application goes offline, all outstanding transactions are terminated. To ensure that any outstanding transactions complete normally, the application can call `flush_wait()` in the `when(offline)` task.

EXAMPLE:

```
boolean nonrecoverable;

...
if (nonrecoverable) {
    go_offline();
}

when (offline)
{
    flush_wait();
    // process shut-down command
}
```

go_unconfigured()

FUNCTION

```
#include <control.h>
void go_unconfigured (void);
```

The `go_unconfigured()` function puts the node into an unconfigured state. It also overwrites all the domain information, which destroys authentication keys as well.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
if (
    (io_in(io_fast_for)==PUSHED) &&
    (io_in(io_set_time)==PUSHED) &&
    (io_in(io_chan_sel_9)==PUSHED))
    go_unconfigured();
// erase network configuration info from this node
```

io_change_init()

Built-In Function

void io_change_init (input_io_object_name [, init_value]);

input_io_object_name specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration.

init_value sets the initial reference value used by the *io_changes* event. If this parameter is omitted, the object's current value is used as the initial reference value.

The *io_change_init()* built-in function initializes the I/O object for the *io_changes* event. If this function is not used, the I/O object's initial reference value defaults to 0.

EXAMPLE:

```
IO_4 input ontime signal;

when(reset)
{
    // Sets comparison value for 'signal' to its current
    // value
    io_change_init(signal);
}
...
when(io_changes(signal) by 10)
{
    ...
}
```

io_edgelog_preload()

Built-In Function

void io_edgelog_preload(value);

value ... A value between 1 and 65,535 defining the maximum value for each period measurement.

The *io_edgelog_preload()* function is optionally used with the *edgelog* I/O object. The *value* parameter defines the maximum value, in units of the clock period, for each period measurement, and may be any value from 1 to 65,535. If the period exceeds the maximum value, the *io_in()* call is terminated.

The default maximum value is 65,535, which provides the maximum timeout condition. By setting a smaller maximum value with this function, a NEURON C program can *shorten* the length of the timeout condition. This function need only be called once, but can be called multiple times to change the maximum value. The function can be called from a *when(reset)* task to automatically reduce the maximum count after every start-up.

If a *preload* value is specified, it must be added to the value returned by *io_in()*. The resulting addition may cause an overflow, but this is normal.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
IO_4 input edgelog elog;

when(reset) {
    io_edgelog_preload(0x4000); // One fourth timeout value (16384)
}
```

io_in()

Built-In Function

return_type io_in (input_io_object_name [, args]);

return_type is the value returned by the function.

input_io_object_name specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration.

args are arguments, which depend on the I/O object type, as described below. Some of these arguments can also appear in the I/O object declaration. If specified in both places, the value of the function argument overrides the declared value for that call only. If the value is not specified in either the function argument or the declaration, the default value is used.

The **io_in()** built-in function reads data from an input object.

The include file *io_types.h* contains optional type definitions for each of the I/O object types. The type names are the I/O object type name followed by “_t”. For example *bit_t* is the type name for a bit I/O object.

io_in_request()

Built-In Function

```
void io_in_request(input_io_object_name, control_value);
```

input_io_object_name Specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration. This built-in function is used only for dualslope I/O objects.

control_value An unsigned long value used to control the length of the first integration period. See the description of the dualslope I/O object for more information.

The **io_in_request()** function is used with a dualslope I/O object. The **io_in_request()** starts the dualslope A/D conversion process.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
IO_4 input dualslope ds;
timer repeating t;
when ~(reset)
{
    t = 5;          // Do a conversion every 5 sec
}
when (timer_expires(t))
{
    io_in_request(ds, 40000);
}
```


The include file `io_types.h` contains optional type definitions for each of the I/O object types. The type names are the I/O object type name followed by “_t”. For example `bit_t` is the type name for a bit I/O object.

The data type of output_value is listed below for each object type:

Object Type	Output Value Type
bit output	unsigned short
bitshift output	unsigned long (also, see below)
byte output	unsigned short
frequency output	unsigned long
muxbus output	unsigned short
neurowire master	(see below)
neurowire master	void (also, see below)
neurowire slave	(see below)
neurowire slave	unsigned short (also, see below)
nibble output	unsigned short
oneshot output	unsigned long
parallel	(see below)
pulsecount output	unsigned long
pulsewidth output	unsigned short or unsigned long
serial output	(see below)
triac output	unsigned long
triggeredcount output	unsigned long

For *bitshift* output objects, the syntax is `io_out (bitshift_output_obj , output_value [, numbits]`;
 `numbits` is the number of bits to be shifted out, from 1 to 127. After 16 bits, zeros are shifted out.

For *neurowire* I/O objects, the syntax is `io_out (neurowire_io_obj, output_value, count)`;
 `output_value` is a pointer to a buffer.
 `count` is the number of bits to be written (from 1 to 255).

Note that calling `io_out()` for a neurowire output object is the same as calling `io_in()`. In either case, data is shifted into the buffer from pin IO_10.

io_preserve_input()

Built-In Function

`void io_preserve_input(input_io_object_name);`

`input_io_object_name`

Specifies the I/O object name which corresponds to `io_object_name` in the I/O declaration. This built-in function is only applicable to input timer/counter I/O objects.

The `io_preserve_input()` function is used with an input timer/counter I/O object. If this function is not called, the firmware will discard the first reading on a timer/counter object after a NEURON CHIP reset (or after a device on the multiplexed timer/counter is selected using the `io_select()` function), since the data may be suspect due to a partial update. Calling the `io_preserve_input()` function prior to the first reading, either by an `io_init()` or implicit `input()` override the discard logic. The `io_preserve_input()` call can be placed in a `when (reset)` clause to preserve the first input value after reset. The call can be used immediately after an `io_select()` call to preserve the first value after select.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
IO_5 input ontime ot1;
IO_6 input ontime ot2;
unsigned long variable1;

when (io_update_occurs(ot1))
{
    variable1 = input_value;
    io_select(ot2);
    io_preserve_input(ot2);
}
```

Appendix C — NEURON C Language Reference

For *bitshift* input objects, the syntax is `io_in` —

`numbits` is the number of bits to be shifted in, from 1 to 127. Only the last 16 bits shifted in will be returned. The unused bits are 0 if fewer than 16 bits are shifted in.

For *neurowire* I/O objects, the syntax is `io_in (neurowire_io_obj, input_value, count)`;
 `input_value` is a pointer to a buffer.
 `count` is the number of bits to be read.

The `io_in()` call has an unsigned short return value signifying the number of bits actually transferred if the object is of type 'neurowire slave'. Else, the return value is void. See the *Driving a Seven Segment Display with the NEURON CHIP Engineering Bulletin* (part no. 005-0014-01) for more information.

For *edgelog* input objects, the syntax is `io_in(edgelog_input_obj, buf, count)`;

`buf` is a pointer to a buffer of unsigned long values.

`count` is the maximum number of values to be read.

The `io_in()` call has an unsigned short return value that is the actual number of edges logged.

For *parallel* I/O objects, the syntax is **io_in** (parallel_obj, input_buffer);

input_buffer is a pointer to the `parallel_io_interface` structure.

For *serial input* objects, the syntax is **io_in** (serial_input_obj, input_buffer, count);

input_buffer is a pointer to a buffer.

For *infrared* input objects, the syntax is **io_in** (ir_input_obj, buf, ct, v1, v2);

buf is a pointer to a buffer.

ct is the maximum number of bits to be read.

v1 is the max period value (an unsigned long). See the I/O object description for more information.

v2 is the threshold value (an unsigned long). See the I/O object description for more information.

The `io_in()` call has an unsigned short return value that is the actual number of bits read.

For *magcard* input objects, the syntax is **io_in** (magcard_input_obj, buf);

buf is a pointer to a 20 byte buffer, which can contain up to 40 hex digits, packed 2 per byte.

The `io_in()` call has a signed short return value that is the actual number of hex digits read. A value of -1 is returned in case of error.

For *muxbus* I/O objects, the syntax is **io_in** (muxbus_io_obj [,addr]);

addr is an optional address to read. Omission of the address will cause the firmware to reread the last address read or written (muxbus is a bi-directional device).

Number of bytes to be read (from 1 to 255).

Built-In Function

io_out (I/O_obj, output_value [, args]);

I/O_obj is the I/O object name, which corresponds to *ct_name* in the I/O declaration.

output_value is the value to be written to the I/O object.

args

are arguments, which depend on the object type, as described below. Some of these arguments can also appear in the object declaration. If specified in both places, the value of the function argument overrides the declared value for that call only. If the value is not specified in either the function argument or the declaration, the default value is used.

The **io_out()** built-in function writes data to an I/O object.

The include file *io_types.h* contains optional type definitions for each of the I/O object types. The type names are the I/O object type name followed by “_t”. For example *bit_t* is the type name for a bit I/O object.

The data type of output_value is listed below for each object type:

Object Type	Output Value Type
bit output	unsigned short
bitshift output	unsigned long (also, see below)
byte output	unsigned short
frequency output	unsigned long
auxbus output	unsigned short
neurowire master	(see below)
neurowire master	void (also, see below)
neurowire slave	(see below)
neurowire slave	unsigned short (also, see below)
nibble output	unsigned short
oneshot output	unsigned long
parallel	(see below)
pulsecount output	unsigned long
pulsewidth output	unsigned short or unsigned long
serial output	(see below)
triac output	unsigned long
triggeredcount output	unsigned long

For *bitshift output* objects, the syntax is **io_out** (bitshift_output_obj , output_value [, numbits]);

numbits is the number of bits to be shifted out, from 1 to 127. After 16 bits, zeros are shifted out.

For *neurowire I/O* objects, the syntax is **io_out** (neurowire_io_obj, output_value, count);

output_value is a pointer to a buffer.

count is the number of bits to be written (from 1 to 255).

Note that calling **io_out()** for a neurowire output object is the same as calling **io_in()**. In either case, data is shifted into the buffer from pin IO_10.

io_preserve_input()

Built-In Function

```
void io_preserve_input(input_io_object_name);
```

input_io_object_name Specifies the I/O object name which corresponds to *io_object_name* in the I/O declaration. This built-in function is only applicable to input timer/counter I/O objects.

The **io_preserve_input()** function is used with an input timer/counter I/O object. If this function is not called, the firmware will discard the first reading on a timer/counter object after a NEURON CHIP reset (or after a device on the multiplexed timer/counter is selected using the **io_select()** function), since the data may be suspect due to a partial update. Calling the **io_preserve_input()** function prior to the first reading, either by an **io_init()** or implicit input, will override the discard logic. The **io_preserve_input()** call can be placed in a **when (reset)** clause to preserve the first input value after reset. The call can be used immediately after an **io_select()** call to preserve the first value after select.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
IO_5 input ontime ot1;
IO_6 input ontime ot2;
unsigned long variable1;

when (io_update_occurs(ot1))
{
    variable1 = input_value;
    io_select(ot2);
    io_preserve_input(ot2);
}
```

For *muxbus* I/O objects, the syntax is `io_out (muxbus_io_obj [,addr],data);`

`output_value` is a pointer to the `parallel_io_interface` structure.

For *serial output* objects, the syntax is `io_out (serial_output_obj, output_value, count);`

`output_value` is a pointer to a buffer.

`count` is the number of bytes to be written (from 1 to 255).

EXAMPLE:

```
boolean value;  
IO_0 output bit d0;  
  
io_out (d0, value);
```

io_out_request()

Built-In Function

`void io_out_request (io_object_name);`

`io_object_name` specifies the I/O object name, which corresponds to *io_object_name* in the parallel I/O declaration.

The `io_out_request()` built-in function sets up the system for an `io_out()` on the specified parallel I/O object. When the system is ready, the `io_out_ready` event becomes TRUE and the `io_out()` function can be used to write data to the parallel port. See Chapter 2 for more information.

EXAMPLE:

```
when (...)  
{  
    io_out_request(io_bus);  
}
```

io_select()

Built-In Function

void **io_select** (input io_object_name [, clock value]);

input io_object_name specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration.

This built-in function is used only for the following timer/counter input objects:

- infrared
- ontime
- period
- pulsecount
- totalcount

clock value optionally specifies a clock, in the range 0 to 7, or a variable name for the clock. This value permanently overrides a clock value specified in the object's declaration.

The clock value option can only be specified for the infrared, ontime, and period objects.

The **io_select()** built-in function selects which of the multiplexed pins is the owner of the timer/counter circuit and optionally specifies a clock for the I/O object. Input to one of the timer/counter circuits can be multiplexed among pins 4 to 7. The other timer/counter input is dedicated to pin 4.

NOTE: **io_select()** automatically discards the first value obtained.

EXAMPLE:

```
IO_5 input ontime pcount1;  
IO_6 input ontime pcount2;  
unsigned long variable1;
```

```
when(io_update_occurs (pcount_1))  
{  
    variable1 = input_value;  
    // select next I/O object  
    io_select(pcount_2);  
}
```

io_set_clock()

Built-In Function

void io_set_clock (io_object_name, clock value);

io_object_name specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration. *This built-in function is used only for timer/counter I/O objects.*

clock value optionally specifies a clock, in the range 0 to 7, or a variable name for the clock. This value overrides a clock value specified in the object's declaration.

The io_set_clock() built-in function allows an application to specify an alternate clock value for any timer/counter object which permits a clock argument in its declaration syntax. The objects are:

dualslope
edgelog
frequency
infrared
oneshot
ontime
period
pulsecount
pulsewidth
triac

The io_set_clock() function can be used to specify a clock for any I/O object, input or output. For multiplexed inputs, use the io_select() function to specify an alternate clock.

Note: When io_set_clock() is used, the I/O object automatically discards the first value obtained

EXAMPLE:

```
IO_1 output pulsecount clock(3)pcout;
```

```
when(...)  
{  
    io_set_clock(pcout, 5);  
    ...  
}
```

io_set_direction()

BUILT-IN FUNCTION

```
typedef enum {IO_DIR_IN=0, IO_DIR_OUT=1} io_direction;
```

```
void io_set_direction (io_object_name, io_direction dir);
```

io_object_name specifies the I/O object name, which corresponds to *io_object_name* in the I/O declaration. ~~This built-in function is used only for timer/counter I/O objects.~~

io_direction dir choose value from the *io_direction* enum shown above.

The **io_set_direction()** function allows the application to change the direction of any bit, nibble or byte type I/O pin at runtime. The "dir" parameter is optional. If not provided, **io_set_direction()** sets the direction based on the direction specified in the declaration of *io_object_name*.

A program can define multiple types of I/O objects for a single pin. When directions conflict and a timer/counter object is defined, the direction of the timer/counter object is used, regardless of the order of definition. However, if the program uses the **io_set_direction()** function for such an object, the direction will be changed as specified.

Any *io_change* events requested for input objects may trigger when the object is redirected as an output. This is because the NEURON CHIP returns the last value output on an output object as the input value. Thus, the user may wish to qualify *io_change* events with flags maintained by the program indicating the current direction of the device.

EXAMPLE :

```
IO_0 output bit b0;
IO_0 input byte byte0;
int read_byte;

io_set_direction(b0, IO_DIR_OUT);
io_out(b0, 0);
io_set_direction(byte0); // Defaults to IO_DIR_IN
read_byte = io_in(byte0);
```

is_bound()

BUILT-IN FUNCTION

boolean **is_bound** (name);

name is either a network variable name or a message tag.

The **is_bound()** built-in function indicates whether the specified network variable or message tag is connected. The function returns TRUE if the network variable or message tag is connected, otherwise it returns FALSE. This function can be used to ensure that transactions are initiated only for connected network variables and message tags.

When an unconnected network variable is updated or a message is sent out on an unconnected message tag, "success" completion events are generated, even though no actual network communication takes place. In this instance, even if the unconnected message is a request and no response is received, the **message_succeeds** and **message_completes** events will be TRUE. Similarly, if a network variable poll is made on an unconnected network variable, no network variable update will occur, although the **nv_update_succeeds** event will be TRUE.

To avoid processing unconnected objects, the program can call **is_bound()** first to ensure that the network variable or message tag is actually connected. In most cases, a program can simply ignore the fact that network variables and message tags are unconnected.

For network variables, **is_bound()** returns TRUE if the network variable selector value is less than 0x3000. For message tags, **is_bound()** returns TRUE if the message tag has a valid address in the address table.

EXAMPLE:

```
network input unsigned temp;
...
// Poll temp if it is bound
if (is_bound(temp)) {
    poll(temp);
}
```

max()**BUILT-IN FUNCTION**

type **max** (*a*, *b*);

The **max()** built-in function compares *a* and *b* and returns the larger value. The result type is determined by the types of *a* and *b*, as shown below.

<i>“Larger Type”</i>	<i>“Smaller Type”</i>	<i>Result</i>
unsigned long	(any)	unsigned long
signed long	signed long unsigned short signed short	signed long
unsigned short	unsigned short signed short	unsigned short
signed short	signed short	signed short

If the result type is unsigned, the comparison is unsigned, else the comparison is signed. Arguments can be cast, which affects the result type. When argument types do not match, the “smaller type” argument is promoted to the “larger type” prior to the operation.

EXAMPLE :

```
int a, b, c;  
long x, y, z;  
  
a = max(b, c);  
x = max(y, z);
```

memcpy()

BUILT-IN FUNCTION

void memcpy (void *dest, void *src, unsigned long len);

The **memcpy()** function copies a block of **len** bytes from **src** to **dest**. It does not return any value. This function cannot be used to copy overlapping areas of memory, or to write into EEPROM memory or network variables. However, similar results can be achieved using struct assignment. The **memcpy()** function can also be used to copy to and from the data fields of the **msg_in**, **resp_in**, **msg_out**, and **resp_out** objects.

EXAMPLE:

```
memcpy(msg_out.data, "Hello World", 11);
```

memset()

BUILT-IN FUNCTION

void memset (void *p, int c, unsigned long len);

The **memset()** function sets the first **len** bytes of the block pointed to by **p** to the character **c**. It does not return any value. This function cannot be used to write into EEPROM memory or network variables.

EXAMPLE:

```
unsigned target[20];  
memset(target, 0, 20);
```

min()

BUILT-IN FUNCTION

type min (*a*, *b*);

The **min()** built-in function compares *a* and *b* and returns the smaller value. The result type is determined by the types of *a* and *b*, as shown above for **max()**.

EXAMPLE:

```
int a, b, c;  
long x, y, z;  
  
a = min(b, c);  
x = min(y, z);
```

msg_alloc()

BUILT-IN FUNCTION

boolean **msg_alloc** (void);

The **msg_alloc()** built-in function allocates a nonpriority buffer for an outgoing message. The function returns **TRUE** if a **msg_out** object can be allocated. The function returns **FALSE** if a **msg_out** object cannot be allocated. When this function returns **FALSE**, a program can continue with other processing, if necessary, rather than waiting for a free message buffer.

EXAMPLE:

```
if (msg_alloc()) {  
    // OK. Build and send message  
}
```

msg_alloc_priority()

BUILT-IN FUNCTION

boolean msg_alloc_priority (void);

The **msg_alloc_priority()** built-in function allocates a priority buffer for an outgoing message. The function returns **TRUE** if a priority **msg_out** object can be allocated. The function returns **FALSE** if a priority **msg_out** object cannot be allocated. When this function returns **FALSE**, a program can continue with other processing, if desired, rather than waiting for a free priority buffer.

EXAMPLE :

```
if (msg_alloc_priority()) {  
    // OK. Build and send message  
}
```

msg_cancel()

BUILT-IN FUNCTION

void msg_cancel (void);

The **msg_cancel()** built-in function cancels the message currently being built and frees the associated buffer, allowing another message to be constructed.

If a message is constructed but not sent before the critical section (for example, a task) is exited, the message is automatically cancelled. This function is used to cancel both priority and nonpriority messages.

EXAMPLE :

```
if (msg_alloc()) {  
    ...  
    if (offline()) {  
        // Requested to go offline  
        msg_cancel();  
    } else {  
        msg_send();  
    }  
}
```

msg_free()

BUILT-IN FUNCTION

void msg_free (void);

The **msg_free()** built-in function frees the **msg_in** object for an incoming message.

EXAMPLE:

```
...
    if (msg_receive()) {
        // Process message
        ...
        msg_free();
    }
...
```

msg_receive()

BUILT-IN FUNCTION

boolean msg_receive (void);

The **msg_receive()** built-in function receives a message into the **msg_in** object. The function returns **TRUE** if a new message is received, otherwise it returns **FALSE**. If no message is pending at the head of the message queue, this function does not wait for one. A program may need to use this function if it receives more than one message in a single task, as in bypass mode. If there already is a “received” message, the earlier one is discarded (that is, its buffer space is freed).

Important Note: Because this function defines a critical section boundary, it should never be used in a when clause expression (i.e. it can be used in a task). Using it in a when clause expression could result in events being processed incorrectly.

The `msg_receive()` function receives all messages in raw form, such that the special events online, offline and wink cannot be used. If the program handles any of these, it should use the `msg_arrives` event, rather than the `msg_receive()` function.

EXAMPLE:

```
...
    if (msg_receive()){
        // Process message
        ...
        msg_free();
    }
...
```

msg_send()

BUILT-IN FUNCTION

`void msg_send (void);`

The `msg_send()` built-in function sends a message using the `msg_out` object.

EXAMPLE:

```
when (io_changes(switch1)to ON)
{
    // Send a message to the motor
    msg_out.tag = motor;
    msg_out.code = MOTOR_ON;
    msg_out.data[0] = ON_FULL;
    msg_send();
}
```

muldiv()

FUNCTION

```
#include <stdlib.h>
```

```
unsigned long muldiv ( unsigned long A, unsigned long B, unsigned long C );
```

The **muldiv()** function permits the computation of $(A*B)/C$ where A, B, and C are all 16-bit values, but the intermediate product of $(A*B)$ is a 32-bit value. Thus, the accuracy of the result is improved. There are two versions of this function: **muldiv()** and **muldivs()**. The **muldiv()** function uses unsigned long arithmetic, while the **muldivs()** function (see below) uses signed long arithmetic.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
#include <stdlib.h>
unsigned long a, b, c, d;
...
d = muldiv(a, b, c);    // d = (a*b)/c
```

muldivs()

FUNCTION

```
#include <stdlib.h>
```

```
signed long muldivs ( signed long A, signed long B, signed long C );
```

The **muldivs()** function permits the computation of $(A*B)/C$ where A, B, and C are all 16-bit values, but the intermediate product of $(A*B)$ is a 32-bit value. Thus, the accuracy of the result is improved. There are two versions of this function: **muldivs()** and **muldiv()**. The **muldivs()** function uses signed long arithmetic, while the **muldiv()** function (see above) uses unsigned long arithmetic.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
#include <stdlib.h>
signed long a, b, c, d;
...
d = muldiv(a, b, c); // d = (a*b)/c
```

node_reset()

FUNCTION

```
#include <control.h>
void node_reset (void);
```

The **node_reset()** function resets the NEURON CHIP hardware. When **node_reset()** is called, all the node's volatile state information is lost. Variables declared with the **eeeprom** or **config** class and the node's network image (which is stored in EEPROM) are preserved across resets and loss of power. The **when(reset)** event evaluates to TRUE after this function is called.

EXAMPLE:

```
#define MAX_ERRORS1 50
#define MAX_ERRORS2 55
int error_count;
...
when(error_count > MAX_ERRORS1)
{
    application_restart();
}
...
when(error_count > MAX_ERRORS2)
{
    node_reset();
}
```

nv_table_index()

BUILT-IN FUNCTION

The **nv_table_index()** built-in function is used to determine the index of a network variable as allocated by the NEURON C compiler. The returned value is in the range 0 to 61.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE :

```
int nv_index;
network output int my_nv;

nv_index = nv_table_index(my_nv);
```

offline_confirm()

FUNCTION

```
#include <control.h>
```

```
void offline_confirm (void);
```

The **offline_confirm()** function allows a node to confirm to a network management tool that the node has finished its clean-up and is now going off line. This function is normally only used in bypass mode (that is, when the offline event is checked for outside of a when clause). If the program is not in bypass mode, use when (offline) rather than offline_confirm().

In bypass mode, when a NEURON CHIP goes off line using offline_confirm(), the program continues to run. It is up to the programmer to determine which events are processed when the NEURON CHIP is offline.

EXAMPLE:

```
...
if (offline){
    // Perform offline cleanup
    ...
    offline_confirm();
}
```

poll()

BUILT-IN FUNCTION

void poll ([network_var]);

network_var is a network variable identifier, array name, or array element. If the parameter is omitted, all input network variables for the node are polled.

The **poll()** built-in function allows a node to request the latest value for one or more of its input network variables. Any input network variable can be polled at any time. If an array name is used, then each element of the array will be polled. An individual element may be polled with use of an array index. When an event expression qualified by an unindexed network variable array name is TRUE, the built-in variable 'nv_array_index' (type short int) may be examined to obtain the element's index to which the event applies. Note that the network variable does not need to be declared as polled.

The new, polled value can be obtained through use of the **nv_update_occurs** event.

If multiple nodes have output network variables connected to the input network variables being polled, multiple updates will be sent in response to the poll. The polling node cannot assume that all updates will be received and processed

independently. This means it is possible for multiple updates to occur before the polling node can process the incoming values. To ensure that all values sent are independently processed, the polling node should declare the input network variable as synchronous input.

EXAMPLE:

```
network input unsigned temp;
...
// Poll temp if it is bound/
if (is_bound(temp)) {
    poll(temp);
}
.
.
.
when (nv_update_occurs(temp))
{
    // New value of temp arrived
}
```

post_events()

FUNCTION

#include <control.h>

void **post_events** (void);

The **post_events()** function defines a boundary of a critical section at which network variable updates and messages are sent and incoming network variable update and message events are posted.

The **post_events()** function is called implicitly by the scheduler at the end of every task body. If the application program calls **post_events()** explicitly, the application should be prepared to handle the special messages online, offline, and wink before checking for any **msg_arrives** event.

The **post_events()** function can also be used to improve network performance. See the *post_events()* *Function* section in Chapter 5 for a more detailed discussion of this feature.

EXAMPLE:

```
boolean still_processing;  
...  
while (still_processing) {  
    post_events();  
    ...  
}
```

power_up()

FUNCTION

```
#include <status.h>
```

```
boolean power_up (void);
```

The **power_up()** function returns TRUE if the last reset resulted from a power up. Any time an application starts up (whether from a reset or from a power up), the **when (reset)** clause becomes TRUE. This function can be used by the application to determine whether the start-up resulted from a power up or not.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE:

```
when (reset)
{
    if (power_up())
        initialize_hardware();
    else {
        // hardware already initialized
        ...
    }
}
```

random()

FUNCTION

```
unsigned int random (void);
```

The **random()** function returns a random number in the range 0 ... 255. The random number is seeded using the unique 48-bit NEURON ID.

EXAMPLE:

```
unsigned value;
...
value = random();
```

refresh_memory()

FUNCTION

```
#include <control.h>
```

```
void refresh_memory (const void * address, unsigned count);
```

The **refresh_memory()** function refreshes a node's EEPROM memory. Refreshing consists of reading every byte of EEPROM, except the NEURON ID, and writing it back. Calling this function periodically (but infrequently) can increase the life of the EEPROM. It can also be used to refresh off-chip EEPROM (NEURON 3150 CHIP).

The count parameter should be as small as possible to avoid locking out network processing for too long a period. Each byte refreshed in a single call uses 20 milliseconds (for nominal EEPROM write times). Under no circumstances should the count exceed 32.

EXAMPLE:

```
#include <control.h>

refresh_memory(0xf008, 2);
// Refreshes two bytes
```

resp_alloc()

BUILT-IN FUNCTION

```
boolean resp_alloc (void);
```

The **resp_alloc()** built-in function allocates an object for an outgoing response. The function returns TRUE if a **resp_out** object can be allocated. The function returns FALSE if a **resp_out** object cannot be allocated.

EXAMPLE:

```
if (resp_alloc()) {
    // OK. Build and send message
}
```

void resp_cancel (void);

The **resp_cancel()** built-in function cancels the response being built and frees the associated **resp_out** object, allowing another response to be constructed.

If a response is constructed but not sent before the critical section (for example, a task) is exited, the response is automatically cancelled. See Chapter 4 for more detailed information.

EXAMPLE :

```
if (resp_alloc()) {  
    ...  
    if (offline()) {  
        // Requested to go offline  
        resp_cancel();  
    } else {  
        resp_send();  
    }  
}
```

resp_free()

BUILT-IN FUNCTION

void resp_free (void);

The **resp_free()** built-in function frees the **resp_in** object for a response. See Chapter 4.

EXAMPLE :

```
...
    if (resp_receive()) {
        // Process message
        ...
        resp_free();
    }
...
```

resp_receive()

BUILT-IN FUNCTION

boolean resp_receive (void);

The **resp_receive()** built-in function receives a response into the **resp_in** object. The function returns **TRUE** if a new response is received, otherwise it returns **FALSE**. If no response is received, this function does not wait for one. A program may need to use this function if it receives more than one response in a single task, as in bypass mode. If there already is a “received” response, the earlier one is discarded (that is, its buffer space is freed). **Important note:** because this function defines a critical section boundary, it should never be used in a when clause (but it can be used within a task). Using it in a when clause could result in events being processed incorrectly. See Chapter 4 for more detailed information.

EXAMPLE:

```
...
    if (resp_receive()) {
        // Process message
        ...
        resp_free();
    }
...
```

resp_send()

BUILT-IN FUNCTION

void resp_send (void);

The **resp_send()** built-in function sends a response using the **resp_out** object. See Chapter 4 for more detailed information.

EXAMPLE:

```
when (msg_arrives(DATA_REQUEST))
{
    int x, y;
    x = msg_in.data(0);
    y = get_response(x);
    resp_out.code = OK;
    // msg_in no longer available
    resp_out.data[0] = y;
    resp_send();
}
```

retrieve_status()

FUNCTION

#include <status.h>

void retrieve_status (status_^{struct}typedef *status_p);

^{typedef} struct status_struct {
 unsigned long status_xmit_errors;
 unsigned long status_transaction_timeouts;
 unsigned long status_rcv_transaction_full;
 unsigned long status_lost_msgs;
 unsigned long status_missed_msgs;
 unsigned status_reset_cause;
 unsigned status_node_state;
 unsigned status_version_number;
 unsigned status_error_log;
 unsigned status_model_number;
} ~~status_struct~~ ^{status_struct};

<code>status_xmit_errors</code>	is a count of the transmission errors that have occurred on the network. A transmission error is detected through a CRC error during packet reception. This error could result from a collision, noisy medium, or excess signal attenuation.								
<code>status_transaction_timeouts</code>	is a count of the timeouts that have occurred in attempting to carry out acknowledged or request/response transactions initiated by the node.								
<code>status_rcv_transaction_full</code>	is the number of times an incoming <code>unackd_rpt</code> , <code>ackd</code> , or request message was lost because there was no more room in the receive transaction database. The size of this database can be set through a pragma at compile time (<code>#pragma receive_trans_count</code>).								
<code>status_lost_msgs</code>	is the number of messages that were addressed to the node that were thrown away because there was no application buffer available for the message. The number of application buffers can be set through a pragma at compile time (<code>#pragma app_buf_in_count</code>).								
<code>status_missed_msgs</code>	is the number of messages that were on the network but could not be received because there was no network buffer available for the message. The number of network buffers can be set through a pragma at compile time (<code>#pragma net_buf_in_count</code>).								
<code>status_reset_cause</code>	is information identifying the source of the most recent reset. The values for this byte are as follows (x = don't care): <table> <tr> <td>powerup reset</td><td>0bxxxxxx1</td></tr> <tr> <td>external reset</td><td>0bxxxxxx10</td></tr> <tr> <td>watchdog timer reset</td><td>0bxxxx1100</td></tr> <tr> <td>software-initiated reset</td><td>0bxxxx10100</td></tr> </table>	powerup reset	0bxxxxxx1	external reset	0bxxxxxx10	watchdog timer reset	0bxxxx1100	software-initiated reset	0bxxxx10100
powerup reset	0bxxxxxx1								
external reset	0bxxxxxx10								
watchdog timer reset	0bxxxx1100								
software-initiated reset	0bxxxx10100								

`status_node_state`

is the state of the node. The states are as follows:

No application	0x01
Unconfigured	0x02
Unconfigured/no application	0x03
Configured/online	0x04
Configured/no application	0x06
Configured/offline	0x0C

`status_version_number`

is the version number, which reflects the NEURON CHIP firmware version. It is also used by the linker to resolve references to system functions in the application.

`status_error_log`

is the most recent error logged by the NEURON CHIP firmware or application. A value of 0 indicates no error. An error in the range of 1 to 127 is an application error and is unique to the application. An error in the range of 128 to 255 is a system error (system errors are documented in Appendix F).

`status_model_number`

is the model number of the NEURON CHIP. The value for this byte is:

0X00 for NEURON 3120 CHIP
0X01 for NEURON 3150 CHIP

The `retrieve_status()` function returns diagnostic status information to the node application. This information is also available to a network management tool over the network, through the network diagnostics status command. The `status_struct` structure, defined in `status.h`, is shown below.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE :

See Chapter 5

The system errors are also available in the include file `NM_ERR.H`.

The values for `status_model_number` should read:

0X08 for NEURON 3120 CHIP
0X00 for NEURON 3150 CHIP

reverse()

BUILT-IN FUNCTION

unsigned int **reverse** (unsigned int a);

The **reverse()** function reverses the bits in *a*.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

EXAMPLE :

```
int value;  
...  
value = 0xE3;  
...  
value = reverse(value);  
// now value is 0xC7
```

scaled_delay()

Built-in Function

void **scaled_delay** (unsigned long count);

count is a value between 0 and 33333. The formula for determining the duration of the delay is based on count and the NEURON CHIP input clock (see below).

The **scaled_delay()** function generates a delay that scales with the NEURON CHIP input clock.

In the formula shown below, *S* is the input clock:

1 = 10 MHz input clock
2 = 5 MHz input clock
4 = 2.5 MHz input clock
8 = 1.25 MHz input clock
16 = 625kHz input clock

The formula for determining the duration of the delay is

$$\text{delay} = (25.2 * \text{count} + 7.2) * S \text{ microseconds}$$

(See also the `delay()` function, which generates a delay which is fixed, independent of the NEURON CHIP input clock.)

EXAMPLE:

```
IO_2 output bit software_one_shot;

io_out(software_one_shot, 1);
//turn it on
scaled_delay(4);
//approx. 108 μsec at 10MHz
io_out(software_one_shot, 0);
//turn it off
```

sleep()

BUILT-IN FUNCTION

void sleep (unsigned int flags [, io_object_name]);

flags

is one or more of the following three flags, or 0 if no flag is specified:

COMM_IGNORE	causes incoming messages to be ignored
PULLUPS_ON	enables all I/O pullup resistors (the service pin pullup is not affected)
TIMERS_OFF	turns off all timers in the program

If two or more flags are used, they must be combined using either the + or the | operator.

io_object_name

specifies an input object for any of pins IO_4 through IO_7. When any I/O transition occurs on this pin, the NEURON CHIP wakes up. If this parameter is not specified, I/O is ignored after the NEURON CHIP goes to sleep.

The **sleep()** built-in function puts the NEURON CHIP in a low-power state. The processors are halted, and the internal oscillator is turned off.

The NEURON CHIP wakes up when:

- A message arrives (unless the **COMM_IGNORE** flag is set);
- The service pin is pressed; or
- An input object transition occurs (if one is specified).

(See also Chapter 5.)

EXAMPLE:

```
IO_6 input bit wakeup;
...
when (flush_completes)
{
    sleep(COMM_IGNORE + TIMERS_OFF, wakeup);
}
```

timers_off()

FUNCTION

```
#include <control.h>
```

```
void timers_off (void);
```

The **timers_off()** function turns off all software timers. This function could be called, for example, before an application goes offline.

EXAMPLE:

```
...
timers_off();
go_offline();
```

update_address()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
```

```
void update_address ( const address_struct * address, int index );
```

The **update_address()** function copies from the structure referenced by the address pointer parameter to the address table entry specified by the index parameter.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE :

```
#include <addrdefs.h>
#include <access.h>
address_struct address_copy;
msg_tag mv_mt;

address_copy = * (access_address(addr_table_index(my_mt)));
// Modify the address_copy here as necessary
update_address(&address_copy, addr_table_index(my_mt));
```

update_config_data()

Function

```
#include <access.h>
```

```
void update_config_data(const config_data_struct *config_data);
```

The **update_config_data()** function copies from the structure referenced by the configuration data pointer parameter to the config_data variable. The config_data variable is declared const, but can be modified via this function.

See the *NEURON 3150 AND 3120 CHIP Data Book* for a description of the data structure, and for further information.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
#include <access.h>
config_data_struct config_data_copy;
config_data_copy = config_data;
// Modify the config_data copy as necessary
update_config_data(&config_data_copy);
```

update_clone_domain()

Function

```
#include <access.h>
```

```
void update_clone_domain(domain_struct *domain, int index);
```

The **update_clone_domain()** function copies from the structure referenced by the domain pointer parameter to the domain table entry specified by the index parameter.

This function differs from **update_domain()** in that it is only used for a cloned node. A cloned node is a node which does not have a unique domain/subnet/node address on the network. Typically, cloned nodes are intended for low-end systems where network management tools are not used for installation. The LONTALK protocol inherently disallows this configuration because nodes reject messages which have the same source address as their own address. The **update_clone_domain()** function enables a node to receive a message with a source address equal to its own address. There are several restrictions when using cloned nodes, see the changes to page 7-31 of the *LONBUILDER User's Guide* earlier in this supplement. More information about cloned nodes can be found in the *NEURON 3150 and 3120 CHIP Data Book*.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

EXAMPLE:

```
#include <access.h>
domain_struct domain_copy;
domain_copy = *(access_domain(0));
//Modify the domain copy as necessary
update_clone_domain(&domain_copy, 0);
```

update_domain()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
```

```
void update_domain ( domain_struct * domain, int index );
```

The `update_domain()` function copies from the structure referenced by the domain pointer parameter to the domain table entry specified by the index parameter.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE:

```
#include <addrdefs.h>
#include <access.h>
domain_struct domain_copy;

domain_copy = *(access_domain(0));
// Modify the domain_copy as necessary
update_domain(&domain_copy, 0);
```

update_nv()

FUNCTION

```
#include <addrdefs.h>
#include <access.h>
```

```
void update_nv ( const nv_struct * nv_entry, int index );
```

The **update_nv()** function copies from the structure referenced by the **nv_entry** pointer parameter to the network variable configuration table entry as specified by the **index** parameter.

If used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

See the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for a description of the data structure, and the *NEURON CHIP-based Installation of LONWORKS Networks Engineering Bulletin* (part no. 005-0022-01) for further information.

EXAMPLE:

```
#include <addrdefs.h>
#include <access.h>
nv_struct nv_copy;
network output int my_nv;

nv_copy = *(access_nv(nv_table_index(my_nv)));
// Modify the nv_copy here as necessary
update_nv(&nv_copy, nv_table_index(my_nv));
```

watchdog_update()

FUNCTION

#include <control.h>

void watchdog_update (void);

The **watchdog_update()** function updates the watchdog timer. The watchdog timer times out in the range of .84 to 1.68 seconds with a 10 MHz NEURON CHIP input clock. The watchdog timer period scales inversely with the input clock frequency. The scheduler updates the watchdog timer before entering each critical section. To ensure that the watchdog timer does not expire, call the **watchdog_update()** function periodically within long tasks (or when in bypass mode). The **post_events()**, **msg_receive()**, and **resp_receive()** functions also update the watchdog timer, as well as the pulsecount output object.

Within long tasks when the scheduler does not run, the watchdog timer may expire, causing a node reset. To prevent the watchdog timer from expiring, an application program can call the **watchdog_update()** function periodically.

EXAMPLE:

```
boolean still_processing;
...
while (still_processing) {
    watchdog_update();
    ...
}
```

C.3 Network Variable Declarations

The complete syntax for declaring a network variable object is one of the following:

network input | output [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* [*= initial-value*];

network input | output [*netvar-modifier*] [*class*] *type* [*connection-info*] *identifier*
[*array-bound*] [*= initializer-list*];

Note: The brackets around the word "array-bound" do not, in this case, indicate an optional field. They are required and must be keyed in by the programmer.

Up to 62 network variables (including array elements) may be declared in a NEURON C program.

Network Variable Modifiers (netvar-modifier)

The following optional modifiers can be included in the declaration of each network variable:

<code>sd_string (<C string constant>)</code>	is used to set a network variable's self-documentation string of up to 1023 bytes. This modifier can only appear once per network variable declaration. If the keyword <code>sync</code> or <code>polled</code> is used (note that these two are mutually exclusive), then the <code>sd_string</code> must follow the other keyword. Note also that the ANSI C feature of concatenated string constants is permitted. Each variable's SD string may have a maximum length of 1023 bytes.
<code>sync synchronized</code>	specifies that all values assigned to this network variable must be propagated, and in their original order.
<code>polled</code>	(used only for output network variables) specifies that the value of the output network variable is to be sent <i>only</i> in response to a poll request from a node that reads this network variable. When this keyword is omitted, the value is propagated over the network every time the variable is assigned a value and also when polled.

Network Variable Classes (class)

Network variables constitute one of the storage classes in NEURON C. They can also be combined with the following classes:

config
const
eeprom
far

Network Variable Types (type)

A network variable can be declared using any of the following types:

- A standard network variable type (SNVT) as described in Chapter 3. Use of a SNVT promotes interoperability. See the *SNVT Guide* for a list of currently defined Standard Network Variable Types.
- Any of the variable types specified in Chapter 1, except for pointers. The types are:

[signed] long int
unsigned long int
signed char
[unsigned] char
[signed] [short] int
unsigned [short] int
enums (int type)

structures and unions of the above types up to 31 bytes long
single-dimension arrays of the above types, up to 62 elements

Note that a structure containing an array of up to 31 bytes in length may be defined as the type of a network variable.

- A typedef. NEURON C provides one predefined typedef:

```
typedef enum {FALSE, TRUE} boolean
```

The user can define other typedefs.

Connection Information (connection-info)

The following optional fields can be included in the declaration of each network variable. Each of these fields is described in the following paragraphs. The fields can be specified in any order. This information can be exported to a network management tool as described in the *Building Application Nodes* section of Chapter 7 of the *LONBUILDER User's Guide* (part no. 078-0002-01A). Note that these connection information assignments can be overridden by a network management tool after a node is installed, unless otherwise specified using the `nonconfig` option, as detailed below.

```
bind_info (  
  [offline]  
  [unackd | unackd_rpt | ackd [(config | nonconfig)]]  
  [authenticated | nonauthenticated [(config | nonconfig)]]  
  [priority | nonpriority [(config | nonconfig)]]  
  [rate_est (const-expr)]  
  [max_rate_est (const-expr)]  
)
```

`offline` is used to signal to the network management tool that a node should be taken offline before an update is made to the network variable. This option is commonly used with a `config` class network variable.

`unackd | unackd_rpt | ackd [(config | nonconfig)]`

selects the LONTALK protocol service to use for updating this network variable. The allowed types are the following:

`unackd` — unacknowledged service; the update is sent once and no acknowledgment is expected.

`unackd_rpt` — unacknowledged service sent multiple times and no acknowledgments are expected.

`ackd` (the default) — acknowledged service with retry; if acknowledgments are not received from all reader nodes before the layer 4 retransmission timer expires, the message will be sent again, up to the retry count.

An unacknowledged (`unackd`) network variable uses minimal network resources to propagate its values to other nodes. As a result, propagation failures are more likely to occur, and failures are not detected by the node. This class

might be used for variables that are updated on a frequent, periodic basis, where loss of an update is not critical, or in cases where the probability of a collision or transmission error is extremely low.

The `unackd_rpt` service might be used when a message is propagated to many nodes. This reduces the network traffic caused by a large number of nodes sending acknowledgements simultaneously. The LONBUILDER network manager (binder) automatically uses the `unackd_rpt` service when a network variable is sent to a group of more than 63 other nodes.

The keyword `config`, the default, indicates that this service type can be changed by a network management tool. This option allows a network management tool to change the service specification at installation time.

The keyword `nonconfig` indicates that this service cannot be changed by a network management tool.

`authenticated | nonauthenticated [(config | nonconfig)]`

specifies whether the network variable update requires use of the authentication feature. (With authentication, the identity of the writer node is verified by all reader nodes.) Abbreviations for authentication are `auth` and `nonauth`. The `config` and `nonconfig` keywords specify whether the authentication designation can be changed by a network management tool.

NOTE: Use only the `ackd` service type with *authenticated*. Do **not** use `unackd` or `unackd_rpt`.

A network variable will be authenticated only if the readers and writers have the *authenticated* keywords specified. However, if only the originator of a network variable update or poll has used the keyword, the variable will not be authenticated (although the update will take place). (In this case, warning messages are issued by the LONBUILDER network manager, since these connections require the authentication protocol to be used needlessly.) See also the *Authentication* section in Chapter 3.

The default is `nonauth (config)`.

the default value. The `config` and `nonconfig` keywords specify whether the priority designation can be changed by a network management tool. The default is `config`. All priority network variables in a node use the same priority time slot since each node is configured to have no more than one priority time slot.

The default is `nonpriority (config)`.

The `priority` keyword affects output or polled input network variables. When a priority network variable is updated, its value will be propagated on the network within a bounded amount of time as long as the node is configured to have a priority slot by a network management tool. (The exact bound is a function of the bit rate and priority.) This is in contrast to a `nonpriority` network variable update, whose delay before propagation is unbounded.

`nonbind` is a message tag which carries no addressing information and does not consume an address table entry. Use `nonbind` for message tags that exclusively use explicit addressing and, therefore, do not require an address table entry.

`rate_est(const-expr)` is the estimated sustained message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780.

`max_rate_est(const-expr)` is the estimated maximum message rate, in tenths of messages per second, that the associated message tag is expected to transmit. The allowable value range is from 0 to 18780.

Note: It may not always be possible to determine `rate_est` and `max_rate_est`. For example, message output rates are often a function of I/O pins whose behavior is known only after a program has been run on a specific node. These values are used by a network management tool to perform network node analysis and are optional.

Although any value in the range 0-18780 may be specified, not all values are used. The values are mapped into encoded values n in the range 0-127. Only the encoded values are stored in the node's SI (self-identification) data. The actual value can be reconstructed from the encoded value. If the encoded value is zero, the actual value is undefined. If the encoded value is in the range 1-127, the actual value is $a=2^{(n/8)-5}$, rounded to the nearest tenth. (The actual value, a , produced by the formula, is in units of messages per second.)

C.4 Timer Declarations

A timer object is declared using one of the following:

mtimer [repeating] *timer_name*;

stimer [repeating] *timer_name*;

mtimer indicates a millisecond timer

stimer indicates a second timer

repeating is an option for the timer to restart itself automatically upon expiration. With this option, accurate timing intervals can be maintained even if the application cannot respond immediately to an expiration event.

timer_name is a user-supplied name for the timer. Assigning a value to this name starts the timer for the specified length of time. The value of a timer object is an unsigned long (0-65,000). A timer that is running or has expired can be started over by assigning a new value to this object. The timer object can be evaluated while the timer is running, and it will indicate the time remaining. Assigning a value of 0 to this timer name turns the timer off. Up to 16 timer objects may be declared in an application.

When a timer expires, the `timer_expires` event becomes TRUE.

An example of declaring a timer and assigning a value to it is:

```
stimer led_timer;

when (reset)
{
    led_timer = 5;  // start timer with value of 5 sec
}
```

The function `timers_off()` can be used to turn off all application timers—for example, before an application goes offline.

See Chapter 2 for a discussion of timer accuracy.

C.5 Built-in Variables and Objects

NEURON C provides six built-in variables and four built-in objects. All of the built-in variables may be viewed using the `eval` function in the NEURON C Debugger. The built-in variables are:

```
config_data
input_is_new
input_value
nv_array_index
nv_in_addr
read_only_data
activate_service_led
```

The built-in objects are:

```
msg_in
msg_out
resp_in
resp_out
```

Following are more detailed descriptions of these built-in elements:

activate_service_led Variable

This variable can be assigned a value by the application program to control the service LED status. Assigning a non-zero value turns the service LED on. Assigning a zero value turns the service LED off. The include file `control.h` contains the definition for the variable as follows:

```
extern system int activate_service_led;
```

This variable is located in RAM space belonging to the NEURON system firmware. Its value is not preserved after a RESET.

There may be a delay of up to one second between the time that the application program sets this variable and the time that its new value is sensed and acted upon by the system firmware. Therefore, attempts to flash the service LED are limited to a period of at least a second.

Example:

```
/*Turn on service LED*/
activate_service_led = TRUE;

/*Turn off service LED*/
activate_service_led = FALSE;
```

Built-in Variables

config_data Variable

This structure defines the hardware and transceiver properties of this node. It is located in EEPROM, and parts of it belong to the application image written during node manufacture, and to the network image written during node installation. The variable is declared in `access.h` as follows (but a program using this variable needs to include both `addrdefs.h` and `access.h`):

```
#define LOCATION_LEN 6
#define NUM_COMM_PARAMS 7

typedef struct {
    unsigned collision_detect : 1; // offset 0x11
    unsigned bit_sync_threshold: 2;
    unsigned filter : 2;
    unsigned hysteresis : 3;
    unsigned : 6; // offset 0x12
    unsigned cd_tail; : 1;
    unsigned cd_preamble : 1;
} direct_param_struct;
```

```

typedef struct {
    unsigned long    channel_id;           // offset 0x00
    char location[LOCATION_LEN];             // offset 0x02
    unsigned comm_clock      : 5; // offset 0x08
    unsigned input_clock     : 3;
    unsigned comm_type       : 3; // offset 0x09
    unsigned comm_pin_dir    : 5;
    unsigned reserved[5];           // offset 0x0A
    unsigned node_priority;         // offset 0x0F
    unsigned channel_priorities;    // offset 0x10
    union {
        unsigned xcvr_params[NUM_COMM_PARAMS];
        direct_param_struct dir_params; // offset 0x11
    } params;
    unsigned non_group_timer : 4; // offset 0x18
    unsigned nm_auth         : 1;
    unsigned preemption_timeout : 3;
} config_data_struct;

```

```
const config_data_struct config_data;
```

The application program may read, but not write this structure using the global declaration `config_data`. The structure is 25 bytes long, and it may be read and written over the network using the read memory and write memory network management messages with `address_mode=2`. For detailed descriptions of the individual fields, see *Section A.6.1* in the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01).

input_is_new Variable

For all timer/counter input objects, the built-in variable `input_is_new` is set to TRUE whenever the `io_in()` call returns an updated value. The type of `input_is_new` is boolean.

input_value Variable

When the `io_changes` or `io_update_occurs` event is evaluated, an implicit call to the `io_in()` function occurs. This call to `io_in()` obtains an input value for the object, which can be accessed using the built-in variable `input_value`. The type of `input_value` is signed long. For example:

```
signed long switch_state;
```

```

when (io_changes(switch_in))
{
    switch_state = input_value;
}

```

Here, the value of the network variable `switch_state` is set to the value of `input_value` (the switch value that was read in the `io_changes` clause).

nv_array_index Variable

When an event expression (one of: `nv_update_occurs`, `nv_update_completes`, `nv_update_fails`, `nv_update_succeeds`) qualified by an unindexed network variable array name is **TRUE**, the built-in variable `nv_array_index` (type `short int`) may be examined to obtain the element's index to which the event applies.

nv_in_addr Variable

The built-in variable `nv_in_addr` may be used to monitor a large number of nodes from a single node. In most instances, the nodes being monitored are all identical, therefore a single connection is acceptable. The connection would likely include many output nodes (the sensors) and a single input node (the monitor). However, the monitor must be able to distinguish between the many sensors. The built-in variable `nv_in_addr` is used to accomplish this.

When an `nv_update_occurs` event is **TRUE**, the `nv_in_addr` built-in variable will be set to contain the addressing information of the sender. The `nv_in_addr` built-in variable is predefined in the NEURON C compiler as follows:

```

typedef struct {
    unsigned domain      : 1 ;
    unsigned flex_domain : 1 ;
    unsigned format      : 6 ;
    struct {
        unsigned subnet;
        unsigned      : 1 ;
        unsigned node  : 7 ;
    } src_addr;
    struct {
        unsigned group;
    } dest_addr;
} nv_in_addr_t;

const nv_in_addr_t nv_in_addr;

```

domain Domain index of the network variable update.

<code>flex_domain</code>	Always 0 for network variable updates.
<code>format</code>	Addressing format used by the network variable update. Contains one of the following values: <ul style="list-style-type: none"> 0 Broadcast 1 Group 2 Subnet/Node 3 NEURON ID 4 Turnaround
<code>src_addr</code>	Source address of the network variable update.
<code>dest_addr</code>	Destination address of the network variable update if group addressing is used as specified by the format field.

When `nv_in_addr` is used in an application, its value will correspond to the last input network variable updated in the application. The contents of `nv_in_addr` will be undefined if no input network variables have been updated. Updates occur when `nv_update_occurs` events are checked or when `post_events()` is called (either explicitly or between tasks) and updates arrive for network variables for which there is no corresponding `nv_update_occurs` check.

The `nv_in_addr` variable is available on 3150-based nodes only. See the *Monitoring Network Variables* section in Chapter 3 for a description of how `nv_in_addr` is used.

read_only_data Variable

This structure is physically located at the start of on-chip EEPROM, at location 0xF000. It defines the node identification, as well as some of the application image parameters. The variable is declared from `access.h` as follows (but a program using this variable needs to include both `addrdefs.h` *and* `access.h`):

```
#define NEURON_ID_LEN    6
#define ID_STR_LEN       8

typedef struct {
    unsigned neuron_id[NEURON_ID_LEN];
    unsigned model_num;
    unsigned                : 4 ;
    unsigned minor_model_num : 4 ;
    const nv_fixed_struct * nv_fixed;
    unsigned read_write_protect : 1 ;
    unsigned                : 1 ;
    unsigned nv_count           : 6 ;
```

```

const snvt_struct * snvt;
unsigned id_string[ID_STR_LEN];
unsigned                : 1 ;
unsigned two_domains    : 1 ;
unsigned                : 6 ;
unsigned address_count  : 4 ;
} read_only_data_struct;
const read_only_data_struct read_only_data;

```

The application program may read, but not write this structure, using the global declaration `read_only_data`. The structure is 23 bytes long, and it may be read and mostly written (except for the first eight bytes) over the network using the read memory and write memory network management messages with `address_mode=1`. It is written during the process of downloading a new application image into the node. For detailed descriptions of the individual fields, see *Section A.1.1* in the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01).

Built-in Objects

msg_in Object

The structure of the incoming message object, as predefined in the NEURON C compiler, is

```

typedef enum {ACKD, UNACKD_RPT, UNACKD, REQUEST} service_type;

struct {
    int code;                // message code
    int len;                 // length of message data
    int data[MAXDATA];       // message data
    boolean authenticated;    // TRUE if authenticated msg has passed challenge
    service_type service;     // service type
    msg_in_addr addr;        // see msg_addr.h include file
} msg_in;

```

See the *Format of an Incoming Message* section in Chapter 4 for a more detailed description of this structure.

msg_out Object

An outgoing message is defined as follows:

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT, UNACKD, REQUEST} service_type;

struct
{
    boolean priority_on;           // TRUE if a priority message (def: FALSE)
    msg_tag tag;                   // message tag (required)
    int code;                      // message code (required)
    int data[MAXDATA];            // message data (default: none)
    boolean authenticated;        // TRUE if to be authenticated (def: FALSE)
    service_type service;         // service type (default: ACKD)
    msg_out_addr dest_addr;       // see include file msg_addr.h (optional)
} msg_out;
```

See the *The msg_out Object Definition* section in Chapter 4 for a more detailed description of this structure.

resp_in Object

The name of the incoming response object is `resp_in`.

The incoming response object is defined as follows:

```
struct
{
    int code;                     // message code
    int len;                      // length of message data
    int data[MAXDATA];           // message data
    resp_in_addr addr;           // see the include file msg_addr.h (optional)
} resp_in;
```

See the *Receiving a Response* section in Chapter 4 for a more detailed description of this structure.

resp_out Object

The name of the outgoing response object is `resp_out`. The response inherits its priority and authentication designation from the request it is replying to. Because the response is returned to the origin of the request, no message tag is necessary.

The outgoing response object is defined as follows:

```
struct
{
    int code;                // message code
    int data[MAXDATA];      // message data
} resp_out;
```

See the *Constructing a Response* section in Chapter 4 for a more detailed description of this structure.

C.6 Reserved Words

The following are all NEURON C reserved words, including those that are standard to the ANSI C language.

ACKD	auto	else
COMM_IGNORE	baud	enum
FALSE	bcd	extern
IO_0	bcd2bin	far
IO_1	bin2bcd	float
IO_10	bind	flush_completes
IO_2	bind_info	for
IO_3	bit	frequency
IO_4	bitshift	goto
IO_5	boolean	if
IO_6	break	input
IO_7	by	input_is_new
IO_8	byte	input_value
IO_9	case	int
IO_DIR_IN	char	invert
PULLUPS_OFF	clock	io_change_init
REQUEST	clockedge	io_changes
TIMERS_OFF	config	io_direction
TRUE	const	io_in
UNACKD	continue	io_out
UNACKD_RPT	ded	io_select
abs	default	io_set_clock
ackd	delay	io_set_direction
addr_table_index	do	io_update_occurs
auth	double	is_bound
authenticated	eeprom	kbaud

leveldetect
long
master
max
max_rate_est
memcpy
memset
min
msg_alloc
msg_alloc_priority
msg_arrives
msg_cancel
msg_completes
msg_fails
msg_free
msg_in
msg_out
msg_receive
msg_send
msg_succeeds
msg_tag
mtimer
mux
network
neurowire
nibble

nonauth
nonauthenticated
nonbind
nonconfig
nonpriority
numbits
nv_array_index
nv_update_completes
nv_update_fails
nv_update_occurs
nv_update_succeeds
offline
oneshot
online
ontime
output
parallel
period
poll
polled
priority
pulsecount
pulsewidth
quad
quadrature
ram

random
rate_est
register
repeating
reset
resp_alloc
resp_arrives
resp_cancel
resp_free
resp_in
resp_out
resp_receive
resp_send
return
reverse
scaled_delay
sd_string
select
serial
service_type
short
signed
sizeof
slave
slave_b
sleep

charge_pump_enable
dualslope
edgelog
fastaccess
infrared
io_edgelog_preload
io_in_request
io_out_request

io_preserve_input
level
magcard
muxbus
nv_in_addr
pulse

sleep_flags	timeout	unackd_rpt
static	timer_expires	union
stimer	to	unsigned
struct	totalcount	void
switch	triac	volatile
sync	triggeredcount	when
synchronized	typedef	while
system	unackd	wink

The programmer cannot use the following names.

<code>_bcd2bin</code>	<code>_leveldetct_input</code>	<code>_msg_len_get</code>
<code>_bin2bcd</code>	<code>_memcpy</code>	<code>_msg_node_set</code>
<code>_bit_input</code>	<code>_memset</code>	<code>_msg_priority_set</code>
<code>_bit_output_hi</code>	<code>_msg_addr_blockget</code>	<code>_msg_receive</code>
<code>_bit_output_lo1</code>	<code>_msg_addr_blockset</code>	<code>_msg_send</code>
<code>_bit_output_lo2</code>	<code>_msg_addr_get</code>	<code>_msg_service_get</code>
<code>_bitshift_input</code>	<code>_msg_addr_set</code>	<code>_msg_service_set</code>
<code>_bitshift_output</code>	<code>_msg_alloc</code>	<code>_msg_succeeds</code>
<code>_bound_mt</code>	<code>_msg_alloc_priority</code>	<code>_msg_tag_set</code>
<code>_bound_nv</code>	<code>_msg_arrives</code>	<code>_neurowire_input</code>
<code>_byte_input</code>	<code>_msg_cancel</code>	<code>_neurowire_output</code>
<code>_byte_output</code>	<code>_msg_auth_get</code>	<code>_nibble_input</code>
<code>_flush_completes</code>	<code>_msg_auth_set</code>	<code>_nibble_output</code>
<code>_frequency_output</code>	<code>_msg_code_arrives</code>	<code>_nv_poll</code>
<code>_init_baud</code>	<code>_msg_code_get</code>	<code>_nv_poll_all</code>
<code>_init_timer_counter1</code>	<code>_msg_code_set</code>	<code>_nv_update_completes</code>
<code>_init_timer_counter2</code>	<code>_msg_completes</code>	<code>_nv_update_fails</code>
<code>_io_change_init</code>	<code>_msg_data_blockget</code>	<code>_nv_update_occurs</code>
<code>_io_changes</code>	<code>_msg_data_blockset</code>	<code>_nv_update_succeeds</code>
<code>_io_changes_by</code>	<code>_msg_data_get</code>	<code>_offline</code>
<code>_io_changes_to</code>	<code>_msg_data_set</code>	<code>_oneshot_output</code>
<code>_io_direction_hi</code>	<code>_msg_domain_get</code>	<code>_online</code>
<code>_io_direction_lo</code>	<code>_msg_domain_set</code>	<code>_parallel_input</code>
<code>_io_input_value</code>	<code>_msg_fails</code>	<code>_parallel_input_ready</code>
<code>_io_set_clock</code>	<code>_msg_format_get</code>	<code>_parallel_output</code>
<code>_io_update_occurs</code>	<code>_msg_free</code>	<code>_parallel_output_ready</code>

`_dualslope_input`
`_dualslope_start`
`_edgelog_input`
`_io_abort_clear`
`_io_set_clock_x2`
`_ir_input`
`_magcard_input`
`_muxbus_read`
`_muxbus_reread`
`_muxbus_rewrite`
`_muxbus_write`

`_parallel_output_request`
`_period_input`
`_pulsecount_output`
`_pulsewidth_output`
`_quadrature_input`
`_resp_alloc`
`_resp_arrives`
`_resp_cancel`

`_resp_code_set`
`_resp_data_blockset`
`_resp_data_set`
`_resp_free`
`_resp_receive`
`_resp_send`
`_select_input_fn`
`_serial_input`

`_serial_output`
`_sleep`
`_timer_expires`
`_timer_expires_any`
`_totalize_input`
`_triac_output`
`_wink`

C.7 I/O Objects

The syntax for specific I/O object types is described in the following sections. Option keywords such as `clockedge`, `baud`, `numbits`, `select`, and `clock` may appear in any order. Each description also lists the data type of `return_value` for `io_in()` and `output_value` for `io_out()`.

The following input/output object types are addressed in this section:

<i>I/O Object Type</i>	<i>Page No.</i>
• Bit Input/Output	C-96
• Bitshift Input/Output	C-98
• Byte Input/Output	C-100
• Frequency Output	C-102
• Leveldetect Input	C-104
• Neurowire Input/Output	C-105
• Nibble Input/Output	C-109
• Oneshot Output	C-111
• Ontime Input	C-113
• Parallel Input/Output	C-115
• Period Input	C-119
• Pulsecount Input	C-121
• Pulsecount Output	C-122
• Pulsewidth Output	C-124
• Quadrature Input	C-127
• Serial Input/Output	C-129
• Totalcount Input	C-131
• Triac Output	C-133
• Triggeredcount Output	C-137

Also see the *NEURON CHIP Input/Output Timing Specification* Engineering Bulletin (part no. 005-0007-01) and the *NEURON 3120 CHIP and NEURON 3150 CHIP Advance Information* document (part no. 005-0018-01) for more information.

dualslope input
edgelog input
infrared input
magcard input
muxbus input/output

Bit Input/Output

DIRECT I/O OBJECT

This I/O object type is used to read or control the logical state of a single pin, where 0 equals low and 1 equals high. For bit input/output, the data type of return value for `io_in()` is an unsigned short. If you wish to enable the NEURON CHIP's built-in pull-up resistors, you should add the statement `#pragma enable_io_pullups` to the NEURON C program (see the *Compiler Directives* section in Chapter 1 for more details).

Syntax

```
pin input bit io_object_name;
```

```
pin output bit io_object_name [=initial_output_level];
```

<i>pin</i>	specifies one of the eleven I/O pins, IO_0 through IO_10. Bit input/output can be used on any pin.
<i>io_object_name</i>	is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.
<i>initial_output_level</i>	is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be 0 or 1. The default is 0.

Usage

```
unsigned int input_value;  
unsigned int output_value;
```

```
input_value = io_in(input_object);  
io_out(output_object, output_value);
```

Bit Input Example

```
IO_1 input bit io_switch_1;    // declares pin 1 as a bit named
                                // io_switch_1

unsigned int switch_on_off;

...

when (reset)
{
    io_change_init(io_switch_1);
}

when (io_changes(io_switch_1))
{
    switch_on_off = input_value;
}
```

Bit Output Example

```
IO_2 output bit io_LED;

unsigned int led_on_off;

...

when(...)
{
    io_out(io_LED, led_on_off);
}
```

This I/O object type is used to shift a data word of up to 16 bits into or out of the NEURON CHIP. Data is clocked in and out by an internally generated clock. For bitshift input/output, the data type of `return_value` for `io_in()`, and the data type of `output_value` for `io_out()`, is an unsigned long.

When using multiple serial I/O devices which have differing baud rates, the following pragma must be used:

```
#pragma enable_multiple_baud
```

This pragma must appear prior to the use of any I/O function (e.g. `io_in`, `io_out`), else the compiler will output an appropriate error message.

Syntax

pin input bitshift [**numbits** (*expr*)] [**clockedge** (+|-)] [**kbaud** (*expr*)]
io_object_name;

pin output bitshift [**numbits** (*expr*)] [**clockedge** (+|-)] [**kbaud** (*expr*)]
io_object_name [=initial_output_level];

pin specifies a NEURON CHIP I/O pin. Bitshift input/output requires adjacent pins. The CLOCK pin is the pin specified, and the DATA pin is the following pin. The pin specification denotes the lower-numbered pin of the pair and can be IO_0 through IO_6, IO_8, or IO_9.

numbits (*expr*) is a constant expression that specifies the number of bits to be shifted in or out. The expression *expr* can evaluate to any number from 1 to 16. The default is 16. Data is shifted in and out with the most significant bit of numbits first. For `io_in()`, only the last 16 bits shifted in will be returned. For `io_out()`, after 16 bits, zeros are shifted out. The number of bits to be shifted can also be specified in the `io_in()` or `io_out()` call (for detailed description of these two calls, see Section C.2 in this appendix). This temporarily overrides the number specified in the device declaration.

clockedge (+ -)	For inputs, this option specifies whether the data is read on the positive-going or negative-going edge of the clock. For outputs, it specifies whether the data is stable on the positive-going or negative-going edge of the clock. The default value is [+].
kbaud (<i>expr</i>)	specifies the baud rate. The expression <i>expr</i> can be 1, 10, or 15. The default is 15 kbaud with a 10 MHz input clock. The baud rate scales proportionally to the input clock.
io_object_name	is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.
initial_output_level	is a constant expression, in ANSI C format for initializers, used to set the state of the clock pin at initialization. The initial state can be 0 or 1; this applies to the clock pin only. The default is 0.

Usage

```

unsigned long input_value;
unsigned long output_value;

input_value = io_in(input_object[, numbits]);
io_out(output_object, output_value[, numbits]);

```

Bitshift Input Example

```

IO_6 input bitshift numbits(8) io_shiftreg_keyboard;
unsigned long keyed_in_data;
...
when (...)
{
    keyed_in_data = io_in(io_shiftreg_keyboard);
}

```

Bitshift Output Example

```
IO_8 output bitshift numbits(5) clockedge(+) io_adc_1_2_control;  
...  
when (...)  
{  
    io_out(io_adc_1_2_control, 0b10010UL);  
}
```

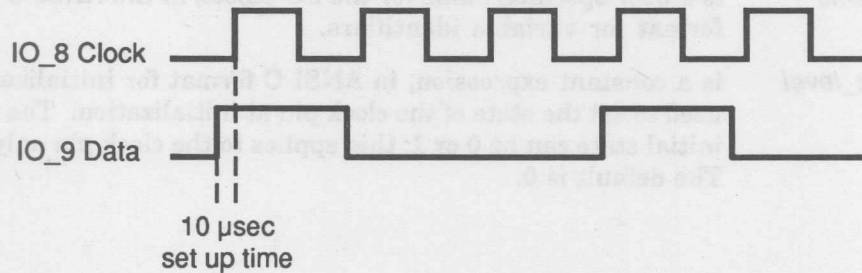


Figure C-1. Bitshift Output

Byte Input/Output

DIRECT I/O OBJECT

This I/O object type is used to read or control eight pins simultaneously. For byte input/output, the data type of return_value for io_in(), and the data type of output_value for io_out(), is an unsigned short.

Syntax

IO_0 Input byte *io_object_name*;

IO_0 output byte *io_object_name* [=initial_output_level];

IO_0 specifies pin IO_0 as the ^{least}~~most~~ significant bit of the byte. Byte input/output uses pins IO_0 through IO_7. The pin specification denotes the lowest numbered pin of the set and must be IO_0.

io_object_name

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial_output_level

is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 255. The default is 0.

Usage

```
unsigned int input_value;  
unsigned int output_value;
```

```
input_value = io_in(input_object);  
io_out(output_object, output_value);
```

Byte Input Example

```
IO_0 input byte io_keyboard;  
unsigned int character;  
...  
when (reset)  
{  
    io_change_init(io_keyboard);  
}  
...  
when (io_changes(io_keyboard))  
{  
    character = input_value;  
}
```

Byte Output Example

```
IO_0 output byte io_LED_display;  
...  
when (...)  
{  
    io_out(io_LED_display, '?');  
}
```

Edgelog Input

Timer/Counter I/O Object

This I/O object type is used to measure a series of both high and low input signal periods on a single input pin, IO_4, in units of the clock period:

$\text{time_on/time_off (ns)} = \text{value_stored} * 2000 * 2^{(\text{clock})} / \text{input_clock (MHz)}$

For edgelog input, the `io_in()` function requires a pointer to a data buffer, into which the series of unsigned long values are stored, and a count argument, which controls the number of values to be stored. The values stored represent the units of clock period between input signal edges, rising or falling. The `io_in()` function returns an unsigned short int that contains the actual number of edge-to-edge periods stored. No input events are associated with the edgelog input object.

During the `io_in()` function call, the measurement process stops whenever the maximum period is exceeded. In this case, the value returned will not be equal to the count argument passed.

The following function is provided specifically for use with the edgelog I/O object:

```
io_edgelog_preload() This function is used to change the maximum  
value for each period measurement. The  
maximum value may range from 1 to 65,535; the  
default value is 65,535.
```

For example, for a 10MHz input clock: an edgelog input object using clock (3) and the default maximum period would yield a 1.6µs resolution and would not overflow until 104.86ms had elapsed. Using a value of 7500 for `io_edgelog_preload()` would result in the `io_in()` function call terminating if 12ms had elapsed with no input edges.

If a preload value is specified, it must be added to the value returned by `io_in()`. The resulting addition may cause an overflow, but this is normal.

This I/O object uses both of the NEURON timer/counters.

If used on a NEURON 3120 CHIP, the edgelog input object is brought in from a system library and placed in on-chip EEPROM when the program is linked.

Syntax

IO_4 [input] edgelog [clock (const_expr)] io_object_name;

IO_4 specifies pin IO_4. This is the input pin for the edgelog input object.

clock (const_expr) specifies a clock rate in the range 0 to 7, where 0 is the fastest and 7 is the slowest. The default clock rate for edgelog input is 2. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10MHz (the values scale with the input clock):

Clock	Input Range and Resolution
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2 (default)	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677sec in steps of 25.6 µs

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

In Figure 4-1, an `io_in()` function call is executed sometime after the IO_4 input signal is sensed as changing to '1', but before it has changed back to '0'. The first period, Period [1], is stored as a value in the array pointed to by the buffer argument. If the `io_in()` function call occurs within the Period [2] time frame, the data for Period [1] is lost.

Individual period measurements may be skipped if the sum of two consecutive periods is less than 104µs (10MHz input clock), regardless of the timer/counter clock setting. The minimum value scales with the input clock.

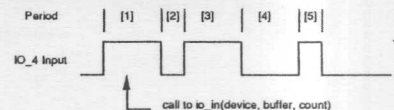


Figure 4-1

If the IO_4 input pin has been at a constant level for longer than the overflow period before the call to `io_in()` is made, the first value stored in the buffer is not the maximum value, but rather the value for the next period.

Usage

```
unsigned int count;  
unsigned long input_buffer[BUFFER_SIZE];  
count = io_in(input_object, input_buffer, count);
```

Frequency Output

TIMER/COUNTER I/O OBJECT

This I/O object type produces a repeating square wave output signal whose period is a function of `output_value` and the selected clock value:

$$\text{period (ns)} = \text{output_value} * 4000 * 2^{(\text{clock}) / \text{input_clock (MHz)}}$$

For frequency output, the data type of `output_value` for `io_out()` is an unsigned long. An `output_value` of 0 forces the output signal to a low state (unless the `invert` keyword is used in the declaration; see below).

Syntax

```
pin output frequency [invert] [clock (const_expr)] io_object_name  
[=initial_output_level];
```

- pin** specifies either pin `IO_0` or `IO_1`.
- invert** normally has no effect other than inverting the output for an output value of 0. The default output for 0 is low.
- clock** (*const_expr*) specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for frequency output is clock 0. The `io_set_clock()` function can be used to change the clock at run-time. The clock values are as follows for an input clock of 10 MHz:

Clock	Period Range
0 (default)	0 to 26.21ms in steps of 400 ns (0-65535)
1	0 to 52.42ms in steps of 800 ns
2	0 to 104.86ms in steps of 1.6 μ s
3	0 to 209.71ms in steps of 3.2 μ s
4	0 to 419.42ms in steps of 6.4 μ s
5	0 to 838.85ms in steps of 12.8 μ s
6	0 to 1.677sec in steps of 25.6 μ s
7	0 to 3.355sec in steps of 51.2 μ s

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Dualslope Input

Timer/Counter I/O Object

This I/O object type is used to control a timer/counter output pin based on a control_value argument and the state of a timer/counter input pin. In this configuration, the NEURON CHIP controls and measures the integration periods of a dual slope integrating A/D converter. When combined with external analog circuitry, the NEURON CHIP performs A/D measurements with 16 bits of resolution for as little as a 13.11ms integration period with a 10MHz input clock (the period scales with the input clock). Faster conversion rates are attainable at the expense of bit resolution. The duration of the first integration period is a function of control_value and the selected clock value:

$$\text{duration (ns)} = \text{control_value} * 2000 * 2^{\text{clock}} / \text{input_clock (MHz)}$$

The value read back by this device reflects the length of the second integration period, and is also in units of the selected clock value:

$$2\text{nd_integration (ns)} = \text{input_value} * 2000 * 2^{\text{clock}} / \text{input_clock (MHz)}$$

A single NEURON timer/counter provides the control output signal and senses a comparator output signal. The control output signal controls an external analog multiplexer which switches between the unknown input voltage and a voltage reference. The timer/counter's input pin is driven by an external comparator which compares an integrator output with a voltage reference, and the count on a low-going transition.

For dualslope input, the data type of control_value for the io_in_request() function is an unsigned long. The return value of the io_in() function is an unsigned long. Both the return value for io_in() and the value stored at input_value is a number biased negatively by the control_value used for the io_in_request() function, and may be corrected by adding the control_value value into it.

For additional information regarding dualslope A/D conversion and the NEURON CHIP, see the *Analog to Digital Conversion with the NEURON CHIP* engineering bulletin (part no. 005-0019-02).

If used on a NEURON 3150 CHIP, the dualslope input object function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

NEURON C Resources

The following functions and events are provided for use with the dualslope input object:

io_in_request() this function starts the first step of the integration process. The control_value argument controls the length of the first integration period.

io_update_occurs this event signals the end of the entire conversion process. The value at input_value now contains the new measurement data.

Syntax

pin input dualslope {mux | ded} [invert] [clock (const_expr)]
io_object_name;

pin specifies a NEURON CHIP I/O pin. Dualslope input can specify pins IO_4 through IO_7.

mux | ded specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin IO_4 is the input pin. The mux keyword assigns the I/O object to the multiplexed timer/counter. The ded keyword assigns the I/O object to the dedicated timer/counter. When the dedicated timer/counter is used the control output pin will be IO_1. When the multiplexed timer/counter is used the control output pin will be IO_0.

invert reverses the logical value of the input pin. Use this keyword if the comparator output is high when the converter is in the idle state.

clock (const_expr) specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for period input is clock 0. The io_set_clock() function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10 MHz (the values scale with the input clock):

Clock	Range and Resolution of Period
0 (default)	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 µs
4	0 to 209.71ms in steps of 3.2 µs
5	0 to 419.42ms in steps of 6.4 µs
6	0 to 838.85ms in steps of 12.8 µs
7	0 to 1.677s in steps of 25.6 µs

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long input_value, control_value;
io_in_request(output_object, control_value);
input_value = io_in(input_object);

stant expression, in ANSI C format for initializers, set the state of the output pin of the I/O object at ation. The initial state is limited to 0 or 1. The is 0.

value);

clock(3) io_alarm;

; // outputs a 3.125 KHz signal at clock(3)

; // outputs a 6.25 KHz signal at clock(3)

// output signal is stopped

Example

```
// Perform a measurement every 500ms
IO_4 input dualslope ded clock(0) dsad_1;
mtimer repeating go_time;
unsigned long raw_ds;

...
when (reset)
{
    go_time = 500;
}

when (timer_expires(go_time))
{
    // Start the first integration period (9ms at 10MHz).
    io_in_request(dsad_1, 45000UL);
}

when (io_update_occurs(dsad_1))
{
    // The value at input_value is biased by the negative value
    // of the control value used. Correct this by adding it back.
    raw_ds = input_value + 45000UL;
}
```

INFRARED INPUT
SEE P C109

Leveldetect Input

DIRECT I/O OBJECT

This I/O object type is used to detect a transition to the logical 0 level of a single pin. The state of the input is latched in hardware every 200 nsec with a 10 MHz input clock (the interval scales at lower input clock speeds), capturing any 0 level input. This event is represented by a 1 value, and is cleared to 0 when read. The leveldetect input object is useful for capturing events of short duration that would otherwise be missed by the bit input object. For leveldetect input, the data type of return value for `io_in()` is an unsigned short. If you wish to enable the NEURON CHIP's built-in pull-up resistors, you should add the statement `#pragma enable_io_pullups` to the NEURON C program (see the *Compiler Directives* section in Chapter 1 for more details).

Syntax

`pin input leveldetect io_object_name;`

pin specifies a NEURON CHIP I/O pin. Leveldetect input can specify one of the pins IO_0 through IO_7.

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

`unsigned int input_value;`

`input_value = io_in(input_object);`

Example

```
IO_6 input leveldetect io_edge_trigger;
...
when (io_changes(io_edge_trigger))
{
    ... // this task will run at each transition to
        // logical 0 level at pin 6
}
```


This I/O object type is used to transfer synchronous serial data from an ISO 7811 track 2 magnetic stripe card reader. The data is presented as a data signal input on pin IO_9, and a clock, or data strobe, signal input on pin IO_8. The data on pin IO_9 is clocked on or just following the falling edge of the clock signal on IO_8, with the least significant bit first.

Data is recognized as a series of 4 bit characters plus an odd parity bit per character. This process begins when the start sentinel (hex B) is recognized, and continues until the end sentinel (hex F) is recognized. No more than 40 characters, including the two sentinels, will be read. The data is stored as packed BCD digits in the buffer space pointed to by the buffer pointer argument to the `io_in()` function with the parity bit stripped, and includes the start and end sentinel characters. This buffer should be 20 bytes long. The data is stored with the first character in the most significant nibble of the first byte in the buffer.

For magcard input, the `io_in()` function requires a pointer to a data buffer, into which the series of BCD pairs are stored. The `io_in()` function returns a signed int that contains the actual number of characters stored.

The parity of each character is checked. The Longitudinal Redundancy Check (LRC) character, which appears just after the end sentinel, is also checked. If either of these tests fail, if more than 40 characters are being clocked in, or if the process aborts due to an input pin event (see below), the `io_in()` function will return the value (-1). The LRC character is not stored.

The magcard object will also use one of I/O pins IO_0 through IO_7 as a timeout/abort pin. Use of this feature is suggested since the `io_in()` function will update the watchdog timer during clock wait states, and could result in a lockup if the card were to stop moving in the middle of the transfer process. If a '1' level is detected on the I/O timeout pin, the `io_in()` function will abort. This input can be a one-shot timer counter output, an RC circuit, or a `-data_valid` signal from the card reader.

If used on a NEURON 3120 CHIP, the magcard input object function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

Syntax

IO_8 [input] magcard timeout (pin_nbr) io_object_name;

IO_8 specifies pin IO_8. Magcard input requires both pins IO_8 and IO_9. Pin IO_8 is the negative-going clock, IO_9 is the serial data input.

timeout(pin_nbr) specifies the timeout signal pin, in the range of IO_0 to IO_7. The NEURON CHIP checks the logic level at this pin whenever it is waiting for either rising or falling edges of the clock. If a logic level of 1 is sensed, the transfer is terminated.

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

A NEURON CHIP with a 10MHz input clock rate can process a bit rate of up to 8334 BPS (at a bit density of 75 bits per inch this is a card speed of 111 inches per second). Most magnetic card stripes contain a 15 bit sequence of zero data at the start of the card, allowing time for the application to start the card reading function. At 8334 bits per second, this period is about 1.8ms. If the scheduler latency is greater than the 1.8ms value, the `io_in()` function will miss the front end of the data stream.

Usage

```
unsigned int count;
unsigned int input_buffer[BUFFER_SIZE];
count = io_in(input_object, input_buffer);
```

Example

```
// In this example I/O pin IO_7 is connected to a -data_valid
// signal which is asserted low as long as a valid clock input is
// being generated by the reader device.
IO_8 input magcard timeout (IO_7) card_data;
// This next object allows monitoring of the -data_valid input
// signal.
IO_7 input bit not_data_valid;
int nibbles_read;
unsigned int in_buffer[20];

when (io_changes(not_data_valid) to 0)
{
    nibbles_read = io_in(card_data, in_buffer);
}
```

contents of the buffer. If the number of bits to be transferred is not a factor of eight as defined by *count*, the last byte transferred into the buffer will contain undefined data bit values in the remaining (unfilled) bit locations.

Output

SERIAL I/O OBJECT

to transfer data using a fully synchronous serial data at the same time as data is shifted out. Neurowire I/O is, such as A/D, D/A converters and display drivers faces that conform with National Semiconductor's SPI interface.

may be configured in master mode or slave mode. The master and slave modes is that the clock signal is an output, and an input for the slave mode.

one or more of the pins IO_0 through IO_7 may be used multiple Neurowire devices to be connected on a 3-wire bus specified as 1, 10 or 20 kbps at a NEURON CHIP input rate scale proportionally with input clock.

one of the pins IO_0 through IO_7 may be designated as a level on the timeout pin causes the Neurowire slave I/O to abort before the specified number of bits has been transferred. The NEURON CHIP watchdog timer from resetting the chip in the requested number of bits are transferred by the external

nodes, up to 255 bits of data may be transferred at a time. application processing until the operation is complete.

at, `io_in()` and `io_out()` require a pointer to the data buffer and an `output_value`. Because Neurowire I/O is bidirectional, the calls are equivalent. Use of either call will initiate a transfer of 8 bits at a time, most significant bit first. The clock is used to clock the data. Data is also then pointed to by `input_value` or `output_value`, most significant bit first. On the rising edge of the clock, overwriting the original data in the buffer.

MUTBUS
PTO

When using multiple serial or Neurowire I/O objects which have differing baud rates, the following pragma must be used:

```
#pragma enable_multiple_baud
```

This pragma must appear prior to the use of any I/O function (e.g. `io_in()`, `io_out()`).

The Neurowire slave object is a library function on a NEURON 3120 CHIP. If it is used on a NEURON 3120 CHIP, this function is brought in from a system library and placed in on-chip EEPROM.

For examples on the use of the Neurowire input/output object, see the following Engineering Bulletins: *Driving a Seven Segment Display with the NEURON CHIP* (part no. 005-0014-01) and *Analog-to-Digital Conversion with the NEURON CHIP* (part no. 005-0019-01).

Syntax

```
IO_8 neurowire master | slave [select (pin_nbr) ] [timeout (pin_nbr) ]  
[kbaud (expr) ] io_object_name;
```

IO_8 specifies pin IO_8. Neurowire requires pins IO_8 through IO_10 and must specify IO_8. The “select” pin must be one of IO_0 through IO_7. Pin IO_8 is the positive-going clock, driven by the NEURON CHIP (or the external master). Pin IO_9 is serial data output and IO_10 is serial data input. Up to 255 bits of data can be transferred at a time.

master specifies that the NEURON CHIP provides the clock on pin IO_8, which is configured as an output pin.

slave specifies that the NEURON CHIP senses the clock on pin IO_8, which is configured as an input pin. The maximum input clock rate is 18kbps, 50/50 duty cycle, with a 10MHz NEURON CHIP input clock. This rate scales proportionally to the input clock.

This I/O object type uses all eleven I/O pins to form an 8 bit address and bi-directional data bus interface. This I/O object uses pins IO_0 through IO_7 for the 8 bit address bus and the 8 bit data bus. Pins IO_8 through IO_10 are control signals which are always driven by the NEURON CHIP:

Pin	Function
IO_0 thru IO_7	Address and bi-directional data
IO_8	C ALS: Address latch strobe, asserted high
IO_9	C WS: Write strobe, asserted low
IO_10	C RS: Read strobe, asserted low

This I/O object provides the capability to build an 8 bit data bus system utilizing an 8 bit address bus. Typically, an 8 bit D-type latch (such as a 74HC573) is connected to the NEURON I/O pins where pins IO_0 through IO_7 are connected to the eight Q inputs. Pin IO_8 is connected to the Latch Enable input. In this configuration, 8 bits of address are latched on the 8 D output pins of the '573 device.

Pins IO_9 and IO_10 are the write and read strobes, normally high.

For muxbus output, the `io_out()` function requires an optional 8 bit address argument, and an 8 bit data argument. If the address argument is provided, the NEURON CHIP will first set pins IO_0 through IO_7 as outputs, then place the address value on these pins, and pulse C ALS from low to high to low. This latches the address into the address data latch device.

If the address is not provided, this step is skipped. The current value latched in the address latch remains unchanged.

The NEURON CHIP then places the data argument value on pins IO_0 through IO_7, and pulses C_WS from high to low to high.

ANALOGIC I/O

For muxbus input the `io_in()` function allows an optional 8 bit address argument only. If this argument is provided, the address is emitted and latched in the same manner as for the `io_out()` function.

Finally, the NEURON CHIP sets pins IO_0 through IO_7 as inputs. It drops C_RS from high to low, inputs the 8 bits of data from pins IO_0 through IO_7, and raises C_RS from low to high. The function then returns the 8 bit data value read.

The address argument is optional as a performance enhancement where a bus device can be repeatedly read from or written to without changing the bus address. It is the application programmer's responsibility to keep track of the current bus address when using this feature.

No events are associated with this I/O object.

If used on a NEURON 3120 CHIP, the muxbus object function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

Syntax

IO_0 muxbus io_object_name;

IO_0 specifies pin IO_0. Muxbus input/output requires all eleven pins and must specify pin IO_0.

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned int data_byte;

```
data_byte = io_in(io_object, address);
data_byte = io_in(io_object);
io_out(io_object, address, data_byte);
io_out(io_object, data_byte);
```

Example

```
IO_0 muxbus local_bus;
...
when ( . . . )
{
    // write two bytes to addresses 0x20 and 0x21, and wait for
    // the data at 0x20 to contain the 0x80 value.
    io_out(local_bus, 0x20, 128);
    io_out(local_bus, 0x21, 1);
    if ((io_in(local_bus, 0x20) & 0x80) == 0)
    {
        // continue to read the same address.
        while ((io_in(local_bus) & 0x80) == 0);
    }
}
```

chip select pin for a Neurowire master. Before transfer, the chip select pin goes low; after the data transfer, the chip select pin goes high. In addition to this with the select keyword, the chip select pin must be declared with an output bit object, unless there is no other pin in use. If no chip select pin is in use, the pin can also be declared as any of the other bit input/output objects for that pin (for example, bit input). For a Neurowire slave.

optional timeout signal pin for a Neurowire master. The range of IO_0, to IO_7. When a timeout signal is generated, the NEURON CHIP will check the logic level at the pin whenever it is waiting for either rising or falling edge of the clock. If a logic level of 1 is sensed, the transfer is terminated. This allows the use of an external timeout signal or an internally generated timeout signal, such as a one-shot output object, to limit the duration of the transfer. The watchdog timer is updated by this object every edge of the clock on pin IO_8. Not used for a Neurowire master.

baud rate for a Neurowire master. The expression evaluates to 1, 10, or 20. The default is 20 kbps with a 1 MHz NEURON CHIP input clock. The baud rate scales proportionally to the input clock. Not used for a Neurowire slave.

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

```
unsigned int count, io_buffer[BUFFER_SIZE];
```

```
io_out(io_object, io_buffer, count);
```

Example

```
IO_8 neurowire master select(IO_2) io_display;
```

```
IO_2 output bit io_display_select = 1;    // active low
```

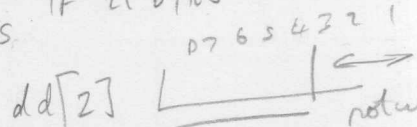
```
unsigned int dd_config = 0x01;    // 8 bits=>display config reg
```

```
unsigned int dd_data[3];    // 24 bits=>display data reg
```

```
when (...)
```

```
{  
    dd_config = 0x01;  
    io_out(io_display, &dd_config, 8);  
    dd_data[0] = 0x80;  
    dd_data[1] = 0xAB;  
    dd_data[2] = 0xCD;  
    io_out(io_display, dd_data, 24);  
}
```

So a transmit < multiply
of 8 move to the
MS bits
a receipt
least sig bits

1st OUT
IF 21 BYTES
dd[2] 

return dd_data[0] is the 1st
if say 6 bits
21 bits

[7
[3
[0] xxx

This I/O object type is used to capture a data stream generated by a class of infrared remote control devices. This class of devices generates a stream of ones and zeros by modulating an infrared emitter for an on and off cycle, each cycle representing either a 'one' or a 'zero'. The period of this on/off cycle determines the data bit value, a shorter cycle implies a 'one', a longer cycle implies a 'zero'.

Typically, the infrared 'on' signal consists of an infrared source modulated at a carrier frequency between 38kHz and 42kHz. An infrared receiver/demodulator is used external to the NEURON CHIP to produce a digital sequence with the carrier removed. Upon execution of the `io_in()` function for the infrared I/O object, the NEURON CHIP measures the cycle times and stores the data bits into a buffer passed to the `io_in()` function.

A timer/counter is used to make the series of cycle time measurements. The resolution of these measurements is in units of the clock period:

$$\text{period (ns)} = \text{measured_value} * 2000 * 2^{\text{clock}} / \text{input_clock (MHz)}$$

For infrared input, the `io_in()` function requires, in addition to the `io_object_name`, four arguments: A pointer to a data buffer in which the series of data bits are stored; a `bit_count` argument, which is the expected number of data bits to be received and stored; a `max_period` argument limiting the range of the timer/counter measurement process; and a threshold argument, representing the half way point, in timer/counter count clocks, between a '0' data period and a '1' data period.

The value returned by the `io_in()` function is the actual number of bits read. If less than `bit_count` bits appear at the input pin, the `io_in()` function waits for the `max_period` period before returning. If more than `bit_count` bits appear at the input pin, the `io_in()` function waits for 'silence' at the input pin before returning. 'Silence' is defined as a lack of input cycles for the `max_period` period. If input cycles persist, the function returns after 256 input cycles occur. This data may be retrieved using the function `rst_bit()` in the extended arithmetic library included on the LONLINK sampler disk.

The `max_period` argument is an unsigned long, and is passed as the negative (two's complement) of the required value. The threshold argument is passed as the `max_period` value plus the required threshold value. See the example below. The edgelog input object type can be used to read inputs from infrared devices that do not conform to the assumptions of the infrared input object type.

If used on a NEURON 3120 CHIP, this infrared input object function is brought in from a system library and placed in on-chip EEPROM when the program is linked.

pin

specifies a NEURON CHIP I/O pin. Nibble input/output requires four adjacent pins. The pin specification denotes the lowest numbered pin of the set and can be IO_0 through

Syntax

`pin input infrared [mux | ded] [invert] [clock (const_expr)] io_object_name;`

`pin` specifies a NEURON CHIP I/O pin. Infrared input can specify pins IO_4 through IO_7.

`mux | ded` specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin IO_4 is the input pin. The `mux` keyword assigns the I/O object to the multiplexed timer/counter. The `ded` keyword assigns the I/O object to the dedicated timer/counter.

`invert` causes the measurement of the cycle period between positive input edges rather than the default, which is between negative input edges.

`clock (const_expr)` specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for period input is clock 6. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10 MHz (the values scale with the input clock):

Clock	Range and Resolution of Period
0 (default)	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μ s
4	0 to 209.71ms in steps of 3.2 μ s
5	0 to 419.42ms in steps of 6.4 μ s
6	0 to 838.85ms in steps of 12.8 μ s
7	0 to 1.677s in steps of 25.6 μ s

`io_object_name` is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Example

```
unsigned int count;
unsigned int input_buffer[BUFFER_SIZE];
unsigned long max_period, threshold;
uint = io_in(input_object, input_buffer, count, max_period, threshold);
```

DIRECT I/O OBJECT

to read or control four adjacent pins simultaneously. The data type of return_value for `io_in()`, and the data type of `io_out()` is an unsigned short. If you wish to enable pull-up resistors, you should add the statement `pull_up = 1` to the NEURON C program (see the *Compiler* for 1 for more details).

`name;`

`name [=initial_output_level];`

4.

The lowest IO pin is defined as the least significant bit of the nibble data.

user-specified name for the I/O object, in the ANSI C format for variable identifiers.

constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state can be from 0 to 15. The default is 0.

Example

This example works with the NEC μ PD1913 encoder chip. This encoder produces a start bit cycle before the actual data stream. During the start bit cycle, the input signal is driven low. This start condition is typical of IR encoders as it allows a receiver/demodulator's AGC circuit time to adjust. It also gives the NEURON CHIP some time to catch this condition from the scheduler, and enter the `io_in()` function. After the start cycle, 32 bits of encoded data appear.

The start cycle is 13ms. The 'zero' cycle is 1.12ms, and the 'one' cycle is 2.24ms. The input clock is 10MHz, and the timer/counter clock is clock (7). This yields a 25.6 μ s timer/counter clock resolution.

The `max_period` parameter is set to cause an overflow at 110% of the start cycle (the timer/counter will count up from this value):

$$65,536 - ((1.0 * 13.0e-3) / 25.6e-6) \text{ or } 64,977.$$

Given the '1' and '0' data periods, the threshold value is:

$$64,977 + (((1.12e-3 + 2.24e-3) / 2) / 25.6e-6) \text{ or } 64,977 + 66$$

This encoder always sends 32 bits, so the count will be 32, and the returned input_buffer will be an array of 4 bytes.

```
// This is the demodulated IR input. Use the non-inverted mode to
// read falling to falling input periods.
IO_4 input infrared ded clock (7) ir_data;
// This object allows the application to monitor the input signal
// before entering the io_in(ir_data) function.
IO_4 input bit ir_data_level;
```

```
unsigned int bits_read;
unsigned int irb[4];
...
when (io_changes(ir_data_level) to 0)
{
    bits_read = io_in(ir_data, irb, 32, 64977UL, 64977UL + 66UL);
    if (bits_read == 32) {
        // So far, a valid data message.
    }
}
```

Nibble Input Example

```
IO_0 input nibble io_column_read;
int column;
...
when (reset)
{
    io_change_init(io_column_read);
}

when (io_changes(io_column_read))
{
    column = input_value;
}
```

Nibble Output Example

```
IO_4 output nibble io_row_write;
...
when (...)
{
    io_out(io_row_write, 0b1000U);
}
```

Oneshot Output

TIMER/COUNTER I/O OBJECT

This I/O object type produces a single output pulse whose duration is a function of `output_value` and the selected clock value:

$\text{duration (ns)} = \text{output_value} * 2000 * 2^{(\text{clock})} / \text{input_clock (MHz)}$;

The oneshot can be retrIGGERED. A call to `io_out()` for a oneshot object will start a new pulse, even if one is currently in progress.

For oneshot output, the data type of `output_value` for `io_out()` is an unsigned long. An `output_value` of 0 forces the output to a low state.

Syntax

`pin output oneshot [invert] [clock (const_expr)] io_object_name
[=initial_output_level];`

- `pin` specifies either pin `IO_0` or `IO_1`.
- `invert` causes the output to be inverted, producing a signal that is normally high with low pulses. The default is normally low with high pulses.
- `clock (const_expr)` specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for oneshot output is clock 7. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10 MHz:

Clock	Oneshot Duration
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.421ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μ s
4	0 to 209.71ms in steps of 3.2 μ s
5	0 to 419.42ms in steps of 6.4 μ s
6	0 to 838.85ms in steps of 12.8 μ s
7 (default)	0 to 1.677sec in steps of 25.6 μ s

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial_output_level is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default is 0.

Usage

unsigned long output_value;

io_out(output_object, output_value);

Example

```
IO_0 output oneshot io_flasher;
unsigned long k = 39062;      // 1 second pulse

mtimer repeating flash_timer;
...
when (...)
{
    flash_timer = 2000;      // start timer, flash every
                             // two seconds
}

when (timer_expires(flash_timer))
{
    io_out(io_flasher, k);    // outputs a 1 second pulse for
                             // k=39062
}
```


Ontime Input

TIMER/COUNTER I/O OBJECT

This I/O object type measures the high or low period of an input signal in units of the clock period:

$$\text{time_on (ns)} = \text{return_value} * 2000 * 2^{(\text{clock})} / \text{input clock (MHz)}$$

For ontime input, the data type of `return_value` for `io_in()` is an unsigned long.

The state of the input pin is latched in hardware every 200 ns with a 10MHz NEURON CHIP input clock (the value scales at lower clock speeds).

Syntax

pin input ontime [mux | ded] [invert] [clock (const_expr)] io_object_name;

pin specifies a NEURON CHIP I/O pin. Ontime input can specify pin IO_4 through IO_7.

mux | ded specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin IO_4 is used as the input pin. The “mux” keyword assigns the I/O object to the multiplexed timer/counter. The “ded” keyword assigns the I/O object to the dedicated timer/counter.

invert causes the measurement of the low period of the input signal. By default, measurement occurs on the high period of the output signal.

clock (const_expr) specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for ontime input is clock 2. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10 MHz:

Clock	Input Range and Resolution
0	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2 (default)	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μ s
4	0 to 209.71ms in steps of 3.2 μ s
5	0 to 419.42ms in steps of 6.4 μ s
6	0 to 838.85ms in steps of 12.8 μ s
7	0 to 1.677sec in steps of 25.6 μ s

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

unsigned long *input_value*;

input_value = **io_in**(*input_object*);

Example

```
IO_4 input ontime ded clock(7) io_gate_time;
unsigned long pulse_duration;
...
when (io_update_occurs(io_gate_time))
{
    pulse_duration = input_value; // measures up to 1.677 seconds
}
```

This I/O object type uses all eleven I/O pins for an 8 bit parallel interface with handshaking. This interface allows data transfer at rates up to 3.3 Mbps. These are the reasons for using a parallel interface:

- To interface a NEURON CHIP to an attached microprocessor or to the bus of a computer system. This interface can use the NEURON CHIP as a communications chip with an existing processor-based system, provide more application performance, or supply more memory. This type of interface is simplified with the LONBUILDER Microprocessor Interface Program (MIP). The MIP moves network variable and explicit message processing to the attached processor.
- For application-level routers and bridges, two NEURON CHIPS are connected back to back across the parallel interface, producing two transceiver interfaces to transport a packet from one channel to the other.

This interface is bidirectional, with the direction (read/write) controlled by the device declared as the master. When using this interface, the NEURON CHIP can be either a master or a slave. The parallel I/O object provides three different configurations of the parallel IO interface: master, slave A, and slave B. Master-slave A connections are typically used for parallel port interfaces and for NEURON CHIP to NEURON CHIP communication. Slave B is typically used for communicating from a microprocessor bus to a NEURON CHIP. Multiple slave B devices can be connected to a single bus. The difference between slave A and slave B concerns the use of one of the three control signals (see the following description of the keywords `slave`, `slave_b`, and `master`).

No other I/O objects can be declared when the parallel I/O object is being used.

For additional information regarding parallel input/output, see the *Parallel I/O Interface to the Neuron Chip* Engineering Bulletin (part no. 005-0021-01).

Neuron C Resources

In order to use the parallel I/O object of the NEURON CHIP, `io_in()` and `io_out()` require a pointer to the `parallel_io_interface` structure defined below:

```
struct parallel_io_interface
{
    unsigned length;           // length of data field
    unsigned data[MAXLENGTH]; // data field
} piofc;
```

The previous structure must be declared, with an appropriate definition of `MAXLENGTH` signifying the largest expected buffer size for any data transfer.

In the case of `io_out()`, `length` is the number of bytes to be transferred out and is set by the application program. In the case of `io_in()`, `length` is the number of bytes to be transferred in. If the incoming length is larger than `length`, then the incoming data stream is flushed, and `length` is set to zero. Otherwise, `length` is set to the number of data bytes read. The `length` field must be set before calling `io_in()` or `io_out()`. The maximum value for the `length` and `MAXLENGTH` fields is 255.

The following functions and events are provided specifically for use with the parallel I/O object:

<code>io_in_ready</code>	this event becomes TRUE whenever a message arrives on the parallel bus that must be read. The application must then call <code>io_in()</code> to retrieve the data.
<code>io_out_request()</code>	this function is used to request an <code>io_out_ready</code> indication for an I/O object. It is up to the application to buffer the data until the <code>io_out_ready</code> event is TRUE. This function acquires the token for the parallel I/O interface.
<code>io_out_ready</code>	this event becomes TRUE whenever the parallel bus is in a state where it can be written to and the <code>io_out_request()</code> function was previously invoked. The application must then call the <code>io_out()</code> function to write the data to the parallel port. This function relinquishes the token for the parallel I/O interface.

NEURON C applications that use the parallel bus in a unidirectional manner may be written (i.e., applications may be written without either an `io_in_ready` or `io_out_ready` when clause).

See the *Parallel I/O Object* section in Chapter 2 for an example which uses the above NEURON C functions and events, as well as for a more general discussion of parallel I/O. Also see the *Parallel I/O Interface to the NEURON CHIP* Engineering Bulletin (part no. 005-0021-01) for more information.

To prevent contention for the data bus, a virtual “write token” is passed back and forth between the master device and the slave device (in both slave A and slave B modes). The master device has the write token initially after a reset. The parallel I/O object declared on a NEURON CHIP automatically manages the write token.

Syntax

IO_0 parallel slave | slave_b | master io_object_name;

IO_0 specifies pin IO_0. Parallel input/output requires all eleven pins and must specify pin IO_0. The pins are used as follows:

Pin	Master	Slave A	Slave B
IO_0 thru IO_7	Data Bus	Data Bus	Data Bus
IO_8	Chip select output	Chip select input	Chip select input
IO_9	RD/~WR output	RD/~WR input	RD/~WR input
IO_10	HANDSHAKE input	HANDSHAKE input	A0 input

slave | slave_b | master specifies slave A, slave B, or master mode. For master and slave A modes, IO_10 is a handshake signal. For slave B mode, IO_10 becomes an address line input, A0, and the handshake signal appears on the data bus on pin IO_0 when A0=1. When A0=0, the data appears on the data bus. This mode is used to allow a NEURON CHIP to reside on a microprocessor bus with the data at one address location and the handshake signal at another.

io_object_name is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

```
struct parallel_io_interface {  
    unsigned int length;  
    unsigned int data[DATA_SIZE];  
} piofc;  
  
io_in(io_object, &piofc);  
io_request(io_object);  
io_out(io_object, &piofc);
```

Example

The following example shows how to use the `io_in_ready` and `io_out_ready` events, in conjunction with the `io_out_request()` function, to handle parallel I/O processing. (See also the description of the *parallel I/O object* in Chapter 2 and the *Parallel I/O Interface to the NEURON CHIP* Engineering Bulletin (part no. 005-0021-01))

```
IO_0 parallel slave s_bus;  
#define DATA_SIZE 255  
struct parallel_io_interface  
{  
    unsigned int length;          // length of data field  
    unsigned int data [DATA_SIZE];  
} piofc;  
  
when (io_in_ready(s_bus))        // ready to input data  
{  
    piofc.length = DATA_SIZE; // number of bytes to read  
    io_in(s_bus &piofc);        // get 10 bytes of incoming data  
}  
  
when (io_out_ready(s_bus))       // ready to output data  
{  
    piofc.length = 10;           // number of bytes to write  
    io_out(s_bus, &piofc);      // output 10 bytes from buffer  
}  
  
when (... )                     // user defined event  
{  
    io_out_request(s_bus);       // post the write transfer request  
}
```


Period Input

TIMER/COUNTER I/O OBJECT

This I/O object type measures the total period, from edge to edge, of an input signal in units of the clock period:

$$\text{period (ns)} = \text{return_value} * 2000 * 2^{(\text{clock})} / \text{input_clock (MHz)}$$

For period input, the data type of `return_value` for `io_in()` is an unsigned long.

The input is latched every 200 ns with a 10 MHz NEURON CHIP input clock. This value scales at lower input clock speeds.

Syntax

`pin input period [mux | ded] [invert] [clock (const_expr)] io_object_name;`

<code>pin</code>	specifies a NEURON CHIP I/O pin. Period input can specify pins IO_4 through IO_7.
<code>mux ded</code>	specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field only applies, and must be used, when pin IO_4 is the input pin. The <code>mux</code> keyword assigns the I/O object to the multiplexed timer/counter. The <code>ded</code> keyword assigns the I/O object to the dedicated timer/counter.
<code>invert</code>	causes the measurement of time between positive edges and typically has no effect. By default, period input measures the time between negative edges.

`clock (const_expr)` specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for period input is clock 0. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for a NEURON CHIP input clock of 10 MHz:

Clock	Range and Resolution of Period
0 (default)	0 to 13.11ms in steps of 200 ns (0-65535)
1	0 to 26.21ms in steps of 400 ns
2	0 to 52.42ms in steps of 800 ns
3	0 to 104.86ms in steps of 1.6 μ s
4	0 to 209.71ms in steps of 3.2 μ s
5	0 to 419.42ms in steps of 6.4 μ s
6	0 to 838.85ms in steps of 12.8 μ s
7	0 to 1.677s in steps of 25.6 μ s

`io_object_name` is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

`unsigned long input_value;`

`input_value = io_in(input_object);`

Example

```
IO_4 input period mux clock(7) io_switch_4;
...
when (io_update_occurs(io_switch_4)) // END OF PERIOD
{
    unsigned short timegap;           // in tenths of a second

    timegap = (unsigned short) (io_in(io_switch_4) / 3906);
                                   // convert to tenths of sec
}
```

Pulsecount Input

TIMER/COUNTER I/O OBJECT

This I/O object type counts the number of input edges at the input pin over a period of 0.8388608 seconds. For pulsecount input, the data type of `return_value` for `io_in()` is an unsigned long.

The input is latched every 200 ns with a 10 MHz NEURON CHIP input clock. This value scales at lower input clock speeds. The value of a pulsecount input object is updated every 0.8388608 seconds and the `io_update_occurs` event becomes TRUE.

Syntax

```
pin input pulsecount [mux | ded] [invert] io_object_name;
```

<i>pin</i>	specifies a NEURON CHIP I/O pin. Pulsecount input can specify pins IO_4 through IO_7.
<i>mux ded</i>	specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin IO_4 is used as the input pin. The <i>mux</i> keyword assigns the I/O object to the multiplexed timer/counter. The <i>ded</i> keyword assigns the I/O object to the dedicated timer/counter.
<i>invert</i>	causes positive edges to be counted and typically has no effect. By default, pulsecount input counts the number of negative input edges.
<i>io_object_name</i>	is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

```
unsigned long input_value;
```

```
input_value = io_in(input_object);
```

Example

```
IO_7 input pulsecount io_total_ticks;
unsigned long k;
...
when (io_update_occurs(io_total_ticks))
{
    k = input_value;    // for up to 65535 ticks per 0.839 seconds
}
```

Pulsecount Output

TIMER/COUNTER I/O OBJECT

This I/O object type produces a sequence of pulses whose period is a function of the clock period:

$$\text{period (ns)} = 256 * 2000 * 2^{(\text{clock}) / \text{input_clock (MHz)}}$$

The `output_value` determines the number of pulses output. When this I/O object is used, the `io_out()` function call does not return until all pulses have been produced. This process ties up the application processor for the duration of the pulsecount.

For pulsecount output, the data type of `output_value` for `io_out()` is an unsigned long. An `output_value` of 0 forces the output signal to its normal state.

Syntax



```
pin output pulsecount [invert] [clock (const_expr)] io_object_name
[=initial_output_level];
```

<i>pin</i>	specifies either pin IO_0 or IO_1.
<i>invert</i>	causes the signal to be inverted, normally high with low pulses. By default, the signal is normally low with high pulses.

clock (*const_expr*)

specifies a clock in the range 1 to 7, where 1 is the fastest clock and 7 is the slowest clock. The default clock for pulsecount output is clock 7. The `io_set_clock()` function can be used to change the clock. (Specifying clock 0 for `io_set_clock()` results in an unspecified number of counts, since this is not a valid clock for pulsecount output.) The periods of the pulses for a NEURON CHIP input clock of 10 MHz are as follows

Clock	Pulse Period (50/50 duty cycle)
1	102.4 μ s
2	204.8 μ s
3	409.6 μ s
4	819.2 μ s
5	1.638 ms
6	3.277 ms
7 (default)	6.554 ms

io_object_name

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial_output_level

is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default is 0.

Usage

unsigned long *output_value*;

io_out(*output_object*, *output_value*);

Example

```
IO_1 output pulsecount io_train_out;
...
when (...)
{
    io_out(io_train_out, 100);    // will produce 100 pulses on
                                // pin 1
}                                // each pulse of period 6.554
                                // milliseconds
```

Pulsewidth Output

TIMER/COUNTER I/O OBJECT

This I/O object type produces a repeating waveform whose duty cycle is a function of `output_value` and whose period is a function of the clock period:

$$\text{pulsewidth (ns)} = \text{output_value} * 2000 * 2^{(\text{clock})} / \text{input_clock (MHz)}$$
$$\text{total_period (ns)} = 256 * 2000 * 2^{(\text{clock})} / \text{input_clock (MHz)}$$

For 8-bit pulsewidth output, the data type of `output_value` for `io_out()` is an unsigned short. An `output_value` of 0 results in a 0% duty cycle. A value of 255 (the maximum value allowed) results in a 100% duty cycle. The duty cycle of the pulse train is $(\text{output_value}/256)$, except when `output_value` is 255; in that case, the duty cycle is 100%.

For 16-bit pulsewidth output, the data type of `output_value` for `io_out()` is an unsigned long. An `output_value` of 0 results in a 0% duty cycle. A value of 65535 (the maximum value allowed) results in a 99.998% duty cycle. The duty cycle of the pulse train is $(\text{output_value}/65536)$.

Syntax

`pin output pulsewidth [short | long] [invert] [clock (const_expr)] io_object_name`
`[=initial_output_level];`

<code>pin</code>	specifies either pin IO_0 or IO_1.
<code>short long</code>	<code>short</code> specifies 8-bit pulsewidth output, <code>long</code> is specified as 16-bit.
<code>invert</code>	causes the output signal to be inverted, normally high for a 0% duty cycle. By default, the output signal is normally low for a 0% duty cycle.

clock (const_expr)

specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default clock for pulsewidth output is clock 0. The `io_set_clock()` function can be used to change the clock. The clock values are as follows for an input clock of 10 MHz:

8-bit Pulsewidth Output

Clock	Control Range
0 (default)	19.53kHz in steps of 200 ns (0-255)
1	9.77kHz in steps of 400 ns
2	4.88kHz in steps of 800 ns
3	2.44kHz in steps of 1.6 μ s
4	1.22kHz in steps of 3.2 μ s
5	610.3Hz in steps of 6.4 μ s
6	305.1Hz in steps of 12.8 μ s
7	152.6Hz in steps of 25.6 μ s

16-bit Pulsewidth Output

Clock	Control Range
0 (default)	76.29Hz in steps of 200 ns (0-65535)
1	38.16Hz in steps of 400 ns
2	19.06Hz in steps of 800 ns
3	9.53Hz in steps of 1.6 μ s
4	4.77Hz in steps of 3.2 μ s
5	2.38Hz in steps of 6.4 μ s
6	1.19Hz in steps of 12.8 μ s
7	0.60Hz in steps of 25.6 μ s

io_object_name

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

initial_output_level

is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state is limited to 0 or 1. The default is 0.

Usage

```
unsigned int output_value;      // for 8-bit output
unsigned long output_value;     // for 16-bit output
```

```
io_out(output_object, output_value);
```

Example

```
IO_1 output pulsewidth clock(7) io_lamp_led;
```

```
mtimer repeating tick_timer;
```

```
unsigned long brightness;
```

```
...
```

```
when (...)
```

```
{
```

```
    tick_timer = 10;          // start clock for fading
```

```
    brightness = 255;         // start brightness for fading
```

```
}
```

```
when (timer_expires(tick_timer))
```

```
{
```

```
    brightness -- 1 ;
```

```
    io_out(io_lamp_led, (short) brightness);
```

```
    if (brightness == 0)
```

```
        tick_timer = 0 ;      // turn off the timer
```

```
}
```

Quadrature Input

TIMER/COUNTER I/O OBJECT

This I/O object type is used to read a shaft or positional encoder input on two adjacent pins. A signed long value is returned from `io_in`, based on the change since the last input. The input is sampled every 200 ns with a 10 MHz NEURON CHIP input clock. This value scales at lower input clock speeds. If you wish to enable the NEURON CHIP's built-in pull-up resistors, you should add the statement `#pragma enable_io_pullups` to the NEURON C program. For more information on quadrature input, see the *Compiler Directives* section in Chapter 1 and the *NEURON CHIP Quadrature Input Function Interface Engineering Bulletin* (part no. 005-0003-01).

Syntax

`pin input quadrature io_object_name;`

pin

specifies a NEURON CHIP I/O pin. Quadrature input requires two adjacent pins. The pin specification denotes the lower-numbered pin of the pair. The pin can be `IO_4` (which uses the dedicated timer/counter) or `IO_6` (which uses the multiplexed timer/counter).

Figure C-2 illustrates the use of the two signal inputs A and B. Both edges of input A are counted. Input B indicates whether input A is moving in a positive or a negative direction.

io_object_name

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

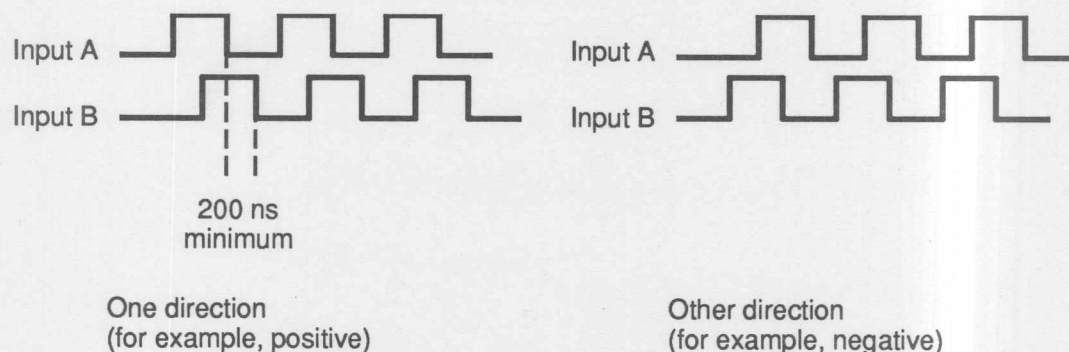


Figure C-2. Quadrature Input

Usage

```
long input_value;
```

```
input_value = io_in(input_object);
```

Example

```
IO_4 input quadrature io_dial;
```

```
long dial_angle = 0;
```

```
when (io_update_occurs(io_dial))
```

```
{
```

```
    dial_angle += input_value;    // integrate angle in software
```

```
}
```

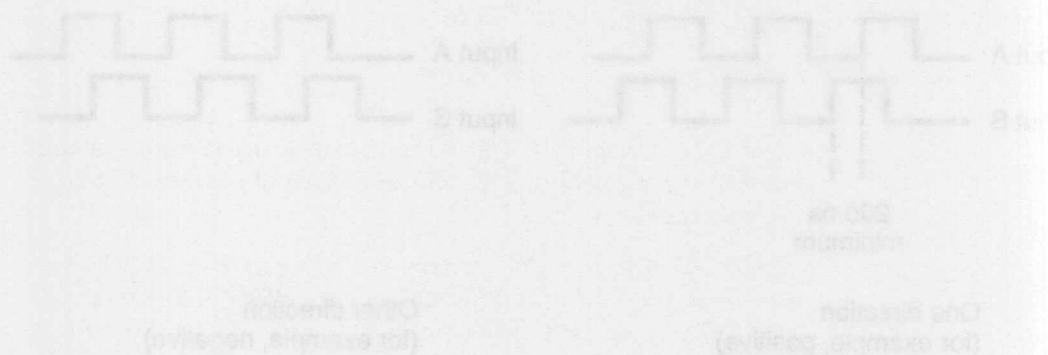


Figure C-2 Quadrature Input

Serial Input/Output

SERIAL I/O OBJECT

This I/O object type is used to transfer data using an asynchronous serial data format, as in RS-232 communications. The format for the transfer is: one start bit, followed by eight data bits (least significant bit first), followed by one stop bit. The input serial I/O object will wait for the start of the data frame to be received for up to the time it would take to receive 20 characters before returning a zero. Input is terminated when either the total count in bytes is received, or the amount of time it would take to receive 20 characters has passed with no data received. The input serial I/O object will stop receiving data on invalid stop bit or parity. At 2400 baud, the input timeout is 83 msec.

When using multiple serial I/O devices which have differing baud rates, the following pragma must be used:

```
#pragma enable_multiple_baud
```

This pragma must appear prior to the use of any I/O function (e.g. `io_in`, `io_out`), else the compiler will output an appropriate error message.

For serial input/output, `io_in()` and `io_out()` require a pointer to the data buffer as the `input_value` and `output_value`. The `io_in()` function returns an unsigned short int that contains the count of the actual number of bytes received. See the *RS-232C Serial Interfacing with the NEURON CHIP* Engineering Bulletin (part no. 005-0008-01) for more information.

Syntax

pin input serial [baud (*const_expr*)] *io_object_name*;

pin output serial [baud (*const_expr*)] *io_object_name*;

pin specifies a NEURON CHIP I/O pin. Serial input requires one pin and must specify IO_8. Serial output also requires one pin and must specify IO_10.

baud (*const_expr*) specifies the bit rate. The expression *expr* can be 600, 1200, 2400, or 4800. The default is 2400 bps with a 10 MHz input clock. The baud rate scales proportionally to the NEURON CHIP input clock.

io_object_name

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

Usage

```
unsigned int count, input_buffer[BUFFER_SIZE], output_buffer[BUFFER_SIZE];
```

```
count = io_in(input_object, input_buffer, count);  
io_out(output_object, output_buffer, count);
```

Serial Input Example

```
IO_8 input serial io_keyboard;  
char in_buffer[20];  
unsigned int num_chars;  
...  
when (...)  
{  
    num_chars = io_in(io_keyboard, in_buffer, 20);  
}
```

Serial Output Example

```
IO_10 output serial io_crt_screen;  
char out_buffer[20];  
...  
when (...)  
{  
    io_out(io_crt_screen, out_buffer, 20);  
}
```


Totalcount Input

TIMER/COUNTER I/O OBJECT

This I/O object type counts the number of input edges at the input pin since the last `io_in()` operation, or since initialization. For totalcount input, the data type of `return_value` for `io_in()` is an unsigned long.

The minimum duration for a high or low input signal for this I/O object is 200 ns with a 10 MHz NEURON CHIP input clock. This value scales at lower input clock speeds.

Syntax

pin input totalcount [mux | ded] [invert] *io_object_name*;

<i>pin</i>	specifies a NEURON CHIP I/O pin. Totalcount input can specify pins IO_4 through IO_7.
<i>mux ded</i>	specifies whether the I/O object is assigned to the multiplexed or dedicated timer/counter. This field is used only when pin IO_4 is used as the input pin. The mux keyword assigns the I/O object to the multiplexed timer/counter. The ded keyword assigns the I/O object to the dedicated timer/counter.
<i>invert</i>	causes positive edges to be counted. By default, totalcount input counts the number of negative input edges.
<i>io_object_name</i>	is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

~~~~~  
`unsigned long input_value;`

`input_value = io_in(input_object);`

### Example

#### ■ Page C-132 Correction

The Totalcount Input example should read:

```
IO_4 input totalcount ded io_event_count;
unsigned long total_num_events = 0;
mtimer repeating t;
...
when (timer_expires(t))
{
    total_num_events += io_in(io_event_count);
    // this sums up all events since initialization-time
}
```

## Triac Output

## TIMER/COUNTER I/O OBJECT

This I/O object type is used to control the delay of an output pulse signal with respect to an input trigger signal. For control of AC circuits using a triac I/O object, the sync input is typically a zero-crossing signal, and the pulse output is the triac trigger signal. The output pulse is 25  $\mu$ s wide, normally low. The pulsewidth is independent of the NEURON CHIP input clock.

Execution of this I/O object type is synchronized with the sync pin input and may not return for up to 10 ms. (The application program could thus be delayed for as long as 10 ms.)

For triac output, the data type of `output_value` for `io_out()` is an unsigned long. An output value of 65535 (the overrange value) assures that no output pulse is generated. This is the equivalent of an OFF state. An output value of 0 is the same as a value of 1, which is used for full ON state (trigger signal immediately follows the sync signal).

$$\text{pulse\_delay (ns)} = \text{output\_value} * 2000 * 2^{(\text{clock})} / \text{input\_clock (MHz)}$$

### Syntax

```
/ pin output triac [pulse|level] sync (pin_nbr) [invert] [clock (const_expr)] [clockedge (+)|(-)|(+)] [clockedge (+)|(-)|(+)] io_object_name;
```

|                       |                                                                                                                                                                                            |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pin</i>            | specifies a NEURON CHIP I/O pin. Triac output can specify pins IO_0 or IO_1. If IO_0 is specified, the sync pin can be IO_4 through IO_7. If IO_1 is specified, the sync pin must be IO_4. |
| <i>sync (pin_nbr)</i> | specifies the sync pin, which is the input trigger signal.                                                                                                                                 |
| <i>invert</i>         | causes the output signal to be inverted, normally high. The default output signal is normally low.                                                                                         |

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>[pulse   level]</i> | <p>specifies whether the output signal produces a 25<math>\mu</math>s pulse at the delay point, or a level, which stays on from the delay point until the next sync input edge.</p> <p>When using the pulse output configuration it should be noted that the output pulse is generated by an internal clock with a constant period of 25.6<math>\mu</math>s (independent of the NEURON CHIP input clock). Since the input sync edge is asynchronous relative to the internal clock there is a jitter associated with the pulse output relative to the input sync edge. This jitter will span a period of 25.6<math>\mu</math>s.</p> |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

`clock (const_expr)`

specifies a clock in the range 0 to 7, where 0 is the fastest clock and 7 is the slowest clock. The default (and recommended) clock for triac output is clock 7. The `io_set_clock()` function can be used to change the clock. Other clock values are as follows for a NEURON CHIP input clock of 10 MHz:

| Clock       | Pulse Delay                                |
|-------------|--------------------------------------------|
| 0           | 0 to 13.11ms in steps of 200 ns ( 0-65535) |
| 1           | 0 to 26.21ms in steps of 400 ns            |
| 2           | 0 to 52.421ms in steps of 800 ns           |
| 3           | 0 to 104.86ms in steps of 1.6 $\mu$ s      |
| 4           | 0 to 209.71ms in steps of 3.2 $\mu$ s      |
| 5           | 0 to 419.42ms in steps of 6.4 $\mu$ s      |
| 6           | 0 to 838.85ms in steps of 12.8 $\mu$ s     |
| 7 (default) | 0 to 1.677sec in steps of 25.6 $\mu$ s     |

`clockedge (+)|(-)|(+/-)`

(+) causes the sync input to be positive-edge sensitive.

(-) (the default) causes the sync input to be negative-edge sensitive.

(+/-) causes the sync input to be both positive- and negative-edge sensitive (valid on the NEURON 3120

CHIP only). Can be used with pulse mode only.

**Note:** the `clockedge (+/-)` option works only with the NEURON 3120 CHIP. When using a NEURON 3150 CHIP, a LONBUILDER emulator, or an SBC, this option cannot be used.

`io_object_name`

is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.

## Usage

`unsigned long output_value;`

`io_out(output_object, output_value);`

### Example 1

```
IO_0 output triac sync (IO_5) io_dimmer_trigger;
...
when (...)
{
    io_out ( io_dimmer_trigger, 325 );    // delay pulse by 8.3
                                          // milliseconds
}
when (...)
{
    io_out ( io_dimmer_trigger, 650 );    // delay pulse by 16.6
                                          // milliseconds
}
when (...)
{
    io_out ( io_dimmer_trigger, 0 ); // full on
}
```

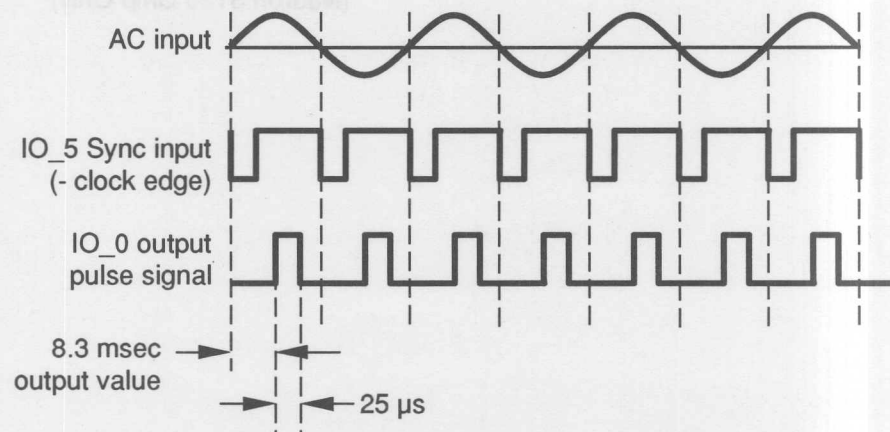


Figure C-3. Triac Output, Example 1  
(NEURON 3120 CHIP Only)

### Example 2

```
IO_1 output triac sync (IO_4) clockedge (+-) io_dimmer_2;  
...  
io_out(io_dimmer_2,325);
```

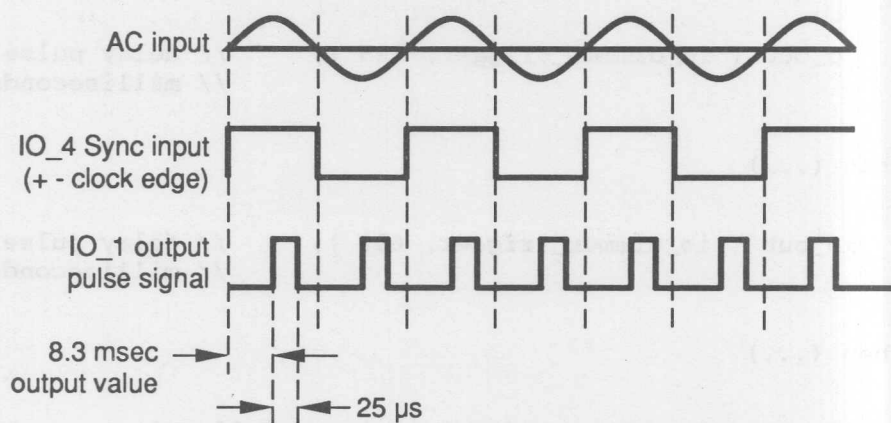


Figure C-4. Triac Output, Example 2  
(Neuron 3120 Chip Only)



## Triggeredcount Output

## Timer/Counter I/O Object

This I/O object type is used to control an output pin to the active state and keep it active until `output_value` negative edges are counted at the input sync pin. After `output_value` edges have counted off, the output pin returns to the low state.

For triggeredcount output, the data type of `output_value` for `io_out()` is an unsigned long. An `output_value` of 0 forces the output signal to an inactive state.

### Syntax

```
pin output triggeredcount sync (pin_nbr) [invert] io_object_name  
[=initial_output_level];
```

|                       |                                                                                                                                                                                                     |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pin</i>            | specifies a NEURON CHIP I/O pin. Triggeredcount output can specify pins IO_0 or IO_1. If IO_0 is specified, the sync pin can be IO_4 through IO_7. If IO_1 is specified, the sync pin must be IO_4. |
| <i>sync (pin_nbr)</i> | specifies the sync pin, which is the counting input signal with low pulses.                                                                                                                         |
| <i>invert</i>         | causes the output signal to be inverted, normally high. By default, the output signal is normally low with high pulses.                                                                             |
| <i>io_object_name</i> | is a user-specified name for the I/O object, in the ANSI C format for variable identifiers.                                                                                                         |

In Figure C-5, an `io_out()` function call is executed with a count argument of 11. After 11 negative edges at the input pin, the output goes low. The delay from the last input edge to the output falling edge is 200 ns or less at a NEURON CHIP input clock of 10 MHz.

*initial\_output\_level* is a constant expression, in ANSI C format for initializers, used to set the state of the output pin of the I/O object at initialization. The initial state may be 0 or 1. The default is 0.

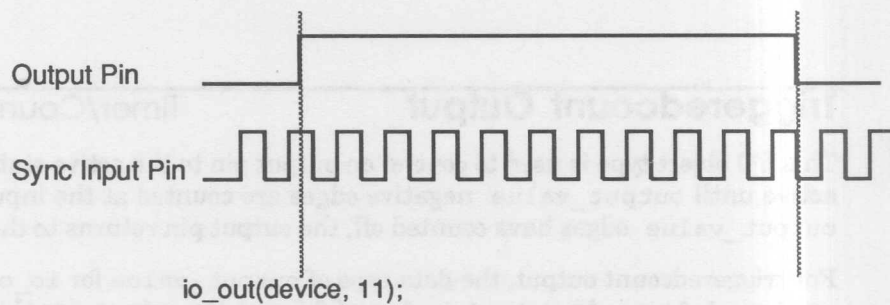


Figure C-5. Triggeredcount Output Object

### Usage

**unsigned long** *output\_value*;

**io\_out**(output\_object, output\_value);

### Example

```
IO_0 output triggeredcount sync (IO_4) io_cascader;
...
when (...)
{
    io_out(io_cascader, 10);    // 1 big output pulse for 10
                              // input pulses
}
```

# Appendix D Implementation Dependencies

This appendix contains explanations of aspects of the NEURON C language that are not specified in the ANSI standard.

---

## Introduction

The American National Standard for the C programming language states (in Appendix F, Section 3), that each C implementation “shall document its behavior in each of the areas listed in this section. The following [aspects of the language] are *implementation-defined*”.

The standard defines the term “implementation-defined” to be “behavior for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document”. Thus, these items are all language definition issues not specified in the ANSI standard. They are also potential portability issues.

Each heading below references the clause in Appendix F (of the ANSI document) and the appropriate section of the ANSI C language standard. Each answer applies to the PC-based, MS-DOS implementation of the NEURON C compiler supplied by Echelon Corporation.

---

### *Translation (F.3.1)*

**Q:** *How is a diagnostic identified? (Sec. 2.1.1.3)*

**A:** Each NEURON C diagnostic consists of at least two lines output to the standard output file. One of these keywords introduces the diagnostic: FYI (For Your Information), Warning, Error, or FATAL. The remainder of the first line consists of the MS-DOS path name of the source or include file to which the diagnostic applies, followed by a line number, and a column number in parentheses.

The second (and possibly subsequent) lines contain the diagnostic. Each of the diagnostic message lines is indented one tab stop.

FYI and warning diagnostics do not prevent the compiler from successfully completing translation. All warning diagnostics should be examined and corrected, however, as

they are likely to indicate programming problems or poor programming practice.

Error diagnostics do prevent the compiler from successfully completing translation. They may also result in masking of other errors; thus the compiler may not be able to locate all errors in a single compilation pass.

FATAL diagnostics prevent the compiler from performing any further translation. These diagnostics result from resource problems (out of memory, disk full, and so on) or from internal checking on the compiler itself. Any diagnostic of the form **\*\*\*TRAP *n*\*\*\***, where *n* is a decimal number, should be reported to Echelon Customer Support for correction.

---

## Environment (F.3.2)

**Q:** *What are the semantics of the arguments to main? (Sec. 2.1.2.2.1)*

**A:** NEURON C places no special meaning on the procedure `main`. The name `main` can be used as any other legal identifier.

**Q:** *What constitutes an interactive device? (Sec. 2.1.2.3)*

**A:** NEURON C defines no interactive devices.

---

## Identifiers (F.3.3)

**Q:** *What is the number of significant initial characters (beyond 31) in an identifier without external linkage? (Sec. 3.1.2)*

**A:** An identifier without external linkage can extend to 256 characters. All characters are significant.

**Q:** *What is the number of significant initial characters (beyond 6) in an identifier with external linkage? (Sec. 3.1.2)*

**A:** There are two forms of external linkage in NEURON C: *traditional external* and *network external*. Traditional external consists of the *extern*, *static*, and *file scope* variables and procedure names. These names are used by the NEURON C linker when linking the program to construct a load image. Names declared with the *extern* or *static* storage classes, or declared at file scope, cannot exceed 63 characters. The compiler does not produce a diagnostic when the length of these names is exceeded. Instead, the compiler simply truncates these names to the maximum length.

The second form of external linkage, *network external*, consists of the names used by the network. These names include names of network variables, names of message tags, and names of *typedefs* used in defining network variables of nonstandard types. The compiler produces an Error diagnostic for each network external name that exceeds 16 characters.

**Q:** *Are case distinctions significant in an identifier with external linkage? (Sec. 3.1.2)*

**A:** Yes, case is significant in an identifier with external linkage, for both forms of external linkage described above.

---

## Characters (F.3.4)

**Q:** *What are the members of the source and execution character sets beyond what the standard explicitly defines? (Sec. 2.2.1)*

**A:** The NEURON C character set uses the basic ASCII character encoding for its source and execution character sets. The NEURON C source character set is the character set as explicitly defined in the standard. The ASCII carriage return character (hex 0D) and the ASCII backspace character (hex 08) are both accepted as white space. The end-of-line



character is the ASCII new-line (hex 0A). Additionally, the NEURON C compiler accepts the remaining basic ASCII printable characters @ (at-sign) and ` (accent-grave) in character constants and string literals.

The NEURON C compiler interprets the ASCII EOT character (hex 04) as an end-of-file marker. Likewise, the character Ctrl-Z (hex 1A), which is the DOS end-of-text-file character, is an end-of-file marker. However, neither of these characters is *required* by the NEURON C compiler.

The execution character set is intended to be basic ASCII (character values 0 .. 127). However, a program written in NEURON C is free to use any interpretation of character values outside the range 0 .. 127.

**Q:** *What are the shift states used for the encoding of multibyte characters? (Sec. 2.2.1.2)*

**A:** NEURON C does not support multibyte characters. Character constants containing more than one character are errors.

**Q:** *What are the number of bits in a character in the execution character set? What is the size of a wide character—that is, the type of wchar\_t? (Sec. 2.2.4.2.1)*

**A:** The execution character set uses an 8-bit representation. The NEURON C compiler does not support wide characters, but the type of a wide character, `wchar_t`, is defined to be unsigned long. (Note that NEURON C defines unsigned long as 16 bits.)

**Q:** *What is the mapping of the members of the source character set (in character constants and string literals) to members of the execution character set? (Sec. 3.1.3.4)*

**A:** The mapping from the source character set to the execution character set is the identity relationship.

**Q:** *What is the value of an integer character constant that contains a character or an escape sequence not represented in the basic execution character set or the extended character set for a wide character constant? (Sec. 3.1.3.4)*

**A:** An integer character constant can only contain characters in the basic execution character set. With escape sequences, character constants can be constructed ranging from 0 to 255, or if signed chars are used, ranging from -128 (\x80) through 127 (\x7F).

**Q:** *What is the value of an integer character constant that contains more than one multibyte character? What is the current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant? (Sec. 3.1.3.4)*

**A:** The NEURON C compiler does not implement multibyte characters.

**Q:** *Does a “plain” char have the same range of values as signed char or unsigned char? (Sec. 3.2.1.1)*

**A:** A “plain” char is identical to an unsigned char.

---

## Integers (F.3.5)

**Q:** *What are the representations and sets of values of the various types of integers? What is the order of bits in a multi-unit integer representation? What is the method of encoding an unsigned integer? What is the method of encoding a signed integer? (Sec. 3.1.2.5)*

**A:** An int implies a short int by default, which is 8 bits in NEURON C. The 8-bit byte is the fundamental unit of storage on the NEURON CHIP. A long int is a 16-bit, or 2-byte, integer representation. The include file <limits.h> contains definitions of the various integer-type ranges. These values are:

|                |           |       |
|----------------|-----------|-------|
| signed short   | -128 ..   | 127   |
| unsigned short | 0 ..      | 255   |
| signed long    | -32768 .. | 32767 |
| unsigned long  | 0 ..      | 65535 |

All unsigned integer values use binary representations. Signed integers use two's complement binary representations. The long int, a multi-unit representation, is stored such that the most significant byte is at the lowest address.

**Q:** *What is the result of converting an integer to a shorter signed integer? What is the result of converting an unsigned integer to a signed integer of equal length, if the signed integer cannot represent the unsigned integer's value? (Sec. 3.2.1.2)*

**A:** Conversion from long to short may result in data loss, depending on the value being converted, since this conversion is performed by discarding the most significant byte of the long integer. If, for example, a long integer containing the value 513 (hex 0201) was converted to a signed short, discarding the most significant byte of the long integer would result in the value 1.

Conversion from an unsigned integer to a signed integer of equal length may result in a negative number. For example, an unsigned short integer may have the value

255 (hex FF), but when converted to a signed short integer, it is then interpreted using two's complement, and the value becomes -1.

Note that the NEURON C compiler produces diagnostic messages when data might be lost as the result of an implicit conversion operation. Explicit conversion will not produce a diagnostic message. As an example, in the code fragment below, the assignment to *x* results in a diagnostic Warning message, but the assignment to *y* does not.

```
int x, y;
x = 285;           // Data is lost, x is assigned 29.
                  // Warning is produced.
y = (int)285;      // Data is lost, y is assigned 29.
                  // No warning is produced.
```

**Q:** *What are the results of bitwise operations on signed integers?* (Sec. 3.3)

**A:** Bitwise operations on signed integers are performed as if the values of the operands were unsigned. The result is interpreted as signed. Thus the result of  $(-2) | 1$  is -1.

**Q:** *What is the sign of the remainder on integer division?* (Sec. 3.3.5)

**A:** The sign of the remainder of an integer division (that is,  $op1 \% op2$ ) is always the same as the sign of *op1*.

**Q:** *What is the result of a right shift of a negative-valued signed integral type?* (Sec. 3.3.7)

**A:** When a negative-valued signed integral type is right-shifted, binary ones are shifted in from the left. Thus, for *int x* and *long int x*,  $(x >> 1)$  is always equal to  $(x/2)$ .

---

## Floating Point (F.3.6)

NEURON C does not support floating point operations. A simple floating point library is available to simplify the use of floating point SNVTs. The library implements a limited set of floating point operations as function calls.

---

## Arrays and Pointers (F.3.7)

**Q:** *What is the type of integer required to hold the maximum size of an array—that is, the type of the size of operator, size\_t? (Secs. 3.3.3.4, 4.1.1)*

**A:** The maximum size of an array (32,767 elements) requires an unsigned long.

**Q:** *What is the result of casting a pointer to an integer, or vice versa? What is the result of casting a pointer to one type to a pointer to another type? (Sec. 3.3.4)*

**A:** The binary representations of pointers and unsigned long integers are the same. Thus, the result of casting a pointer to an integer is the same as casting an unsigned long to an int. Integers cast to pointer undergo the same conversions as integers cast to unsigned long.

All pointer representations are interchangeable. Thus, no conversion results from casting a pointer to one type to a pointer to another type, and the use of such a pointer produces the expected results.

**Q:** *What is the type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff\_t? (Secs. 3.3.6, 4.1.1)*

**A:** The result of subtraction between two pointers is a [signed] long.



---

## Registers (F.3.8)

**Q:** *What is the extent to which objects are actually placed in registers by use of the register storage class specifier? (Sec. 3.5.1)*

**A:** The NEURON CHIP uses a stack-based architecture. Since this architecture has no general-purpose registers, the compiler ignores the register storage class. The compiler also produces a Warning diagnostic whenever the register class is used.

---

## Structures, Unions, Enumerations, and Bit-Fields (F.3.9)

**Q:** *What are the consequences of accessing a member of a union object with a member of a different type? (Sec. 3.3.2.3)*

**A:** Union members of different types overlay each other at the same offsets within a union. Thus, the consequences of accessing a pointer as a long or as an unsigned long, or vice versa, are the same as casting the member. Likewise, the consequences of accessing an int, or char, or short, as another typed member from the same list is the same as casting the member. Accessing a long or pointer as a short will result in the value of the most significant byte. Accessing a short as a long will result in reading or changing an unused byte (the least significant byte of the long), and the most significant byte of the long mapping to the short.

**Q:** *What is the padding and alignment of members of structures? (Sec. 3.5.2.1)*

**A:** Because the architecture of the NEURON CHIP is byte aligned, no padding is needed or performed between members of structures in NEURON C.



**Q:** *Is a “plain” int bit-field treated as a signed int bit-field or as an unsigned int bit-field? (Sec. 3.5.2.1)*

**A:** A “plain” int bit-field is treated as a signed int bit-field. Use of *unsigned* bit-fields is recommended, unless a sign is needed, since *unsigned* bit-fields are more efficient in runtime and code space.

**Q:** *What is the order of allocation of bit-fields within a unit? (Sec. 3.5.2.1)*

**A:** Bit-fields are allocated from high-order bit to low-order bit within a byte.

**Q:** *Can a bit-field straddle a storage-unit boundary? (Sec. 3.5.2.1)*

**A:** No. A bit-field cannot straddle a byte boundary. Therefore, the largest bit-field is 8 bits.

**Q:** *What is the integer type chosen to represent the values of an enumeration type? (Sec. 3.5.2.2)*

**A:** The integer type `int` is used to represent the values of an enumeration type. Thus, the valid range of enumerator values is -128 ... 127.

---

## Qualifiers (F.3.10)

**Q:** *What constitutes an access to an object that has volatile-qualified type? (Sec. 3.5.5.3)*

**A:** NEURON C does not support volatile-qualified type.

---

## Declarators (F.3.11)

**Q:** *What is the maximum number of declarators that can modify an arithmetic, structure, or union type? (Sec. 3.5.4)*

**A:** There is no limit to the maximum number of declarators that modify any type. The limit is determined at run-time by the amount of heap memory and stack space available to the compiler.

---

## Statements (F.3.12)

**Q:** *What is the maximum number of case values in a switch statement? (Sec. 3.6.4.2)*

**A:** The NEURON C switch statement will only accept an int expression for the switch value. Since no two case labels in a switch statement can have the same value, there are only 256 choices permitted. NEURON C will accept all 256 different case values for a single switch statement.

---

## Preprocessing Directives (F.3.13)

**Q:** *Does the value of a single-character constant in a constant expression that controls conditional inclusion match the value of the same character constant in the execution character set? Can such a character constant have a negative value? (Sec. 3.8.1)*

**A:** Yes, and yes.

**Q:** *What is the method for locating includable source files?*  
(Sec. 3.8.2)

**A:** The normal include directive should use the quoted form. To access the system include files, the directive should use the bracketed form.

#### The quoted form

```
#include "[drive:][pathname\]filename.ext"
```

causes the compiler to use the filename as it stands if it is absolute or drive-relative (as defined in MS-DOS).

Otherwise, the compiler will first search the working directory (using a relative pathname if one is supplied). Next it searches each of the directories specified in the Include Directories field of the LONBUILDER Project Configuration screen.

#### The bracketed form

```
#include <filename.ext>
```

searches the working directory and then the include subdirectory within the LONBUILDER system directory for system include files (such as <limits.h> and <stddef.h>). (The search for include files specified by the bracketed form is unaffected by the directories specified in the Include Directories field of the LONBUILDER Project Configuration screen.) The LONBUILDER system directory is the one in which the LONBUILDER software is installed; the default system directory is \lb.

**Q:** *What is the support of quoted names for includable source files?* (Sec. 3.8.2)

**A:** The quoted names in the #include directive can be any valid MS-DOS filename, with absolute, drive-relative, or relative pathname, if any.

**Q:** *What is the mapping of source file character sequences in the #include directive? (Sec. 3.8.2)*

**A:** The source file character sequences can be upper or lower case. Any valid MS-DOS filename character can be used. Case is not significant.

**Q:** *What is the behavior of each recognized #pragma directive? (Sec. 3.8.6)*

**A:** The #pragma directives are documented in Chapters 1 and 6 of this manual.

**Q:** *What are the definitions for \_\_DATE\_\_ and \_\_TIME\_\_ when respectively, the date and time of the translation are not available? (Sec. 3.8.8)*

**A:** NEURON C does not support the \_\_DATE\_\_ and \_\_TIME\_\_ macros.

---

## Library Functions (F.3.14)

**Q:** *What is the NULL pointer constant to which the macro NULL expands? (Sec. 4.1.5)*

**A:** The NULL pointer constant is defined to be 0 in the file <stddef.h>.

NEURON C is a “freestanding implementation.” This means that NEURON C does not include a Standard C library as part of the implementation.

# Appendix E

## Compiler Error Messages

This appendix lists NEURON C compiler error messages and offers suggestions on how to correct the problems.

---

## Introduction

The diagnostic messages produced by NEURON C are of four levels:

- *FYI (For Your Information) messages* are intended simply to provide information.
- *Warning messages* are not severe enough to prevent successful compilation, but they should each be examined, and corrected if appropriate. Any code that is flagged by the compiler with a warning diagnostic should be viewed as a potential programming error, or at least as poor programming practice.
- *Error diagnostics* result from code constructs that cannot be interpreted by the compiler in any way that would produce compilable code, or indicate situations that are expressly prohibited by either the ANSI C language standard, or by NEURON C. A compilation with one or more error diagnostics does not produce any executable files.
- *FATAL diagnostics* prevent the compiler from performing any further translation. These diagnostics result from resource problems (out of memory, disk full, and so on) or from internal checking on the compiler itself. Any diagnostic of the form **\*\*\*TRAP *n*\*\*\***, where *n* is a decimal number, should be reported to Echelon Customer Support.

The following discussions interpret each FYI, Warning, or Error level diagnostic produced by the NEURON C Compiler, and briefly explain how to correct the problem. Error messages appear in *italic* type. The explanation of how to correct the problem appears in Roman type.



---

## Compiler Error Messages

---

Compiler error messages appear in alphabetical order. For additional information concerning topics addressed by these messages, see the index and the remainder of this guide.

---

### A,B

#### ***A msg\_tag declaration is not permitted if micro\_interface***

Once the `#pragma micro_interface` appears, the program cannot declare any network variables or message tags.

#### ***Array in struct or union must have bounds***

##### ***Array must have bound***

Use of the array type in a declaration must include a constant expression which is the array bound. The only time this bound may be omitted is in the declaration of a function parameter. In this case, use of the bound is ignored, and the parameter is actually treated as a pointer.

#### ***Attempt to divide by the constant zero (0)***

The compiler detects attempts to divide by the constant zero, even if this constant is a result of constant folding, or is "hidden" in a `#define` directive, or an enum, etc.

#### ***Authenticated network variables require 'ackd' service type***

Some of the options in the `bind_info` declaration modifier only apply to network variables, some only apply to output network variables, and some only apply to message tags.

#### ***Bad I/O modifier for this I/O object type***

Several of the I/O object declarations permit or require modifiers, like `mux`, `ded`, `sync`, and so on. These are permitted or required on a per I/O object-type basis. At most one of each type of modifier is permitted in a single declaration.

### ***Bad type for conditional expression***

### ***Bad type for operator***

These error diagnostics result from combining expressions of conflicting types, such as assigning an int to a pointer, or a pointer of one type to a pointer to another type, or in using objects that have no value (such as message tags or I/O object names) in expressions.

However, note that casting should be avoided if possible, as it is often poor programming practice.

### ***Base type of network variable is too large***

A network variable array element, struct, or union is limited to 31 bytes.

### ***Bitfield size cannot be 0 unless unnamed***

### ***Bitfield size must be from 0 to 8 bits***

A bitfield must fit inside an int type. Since in NEURON C an int is 8 bits, the bitfield is also limited to 8 bits. Note that in ANSI C a bitfield size of 0 bits is legal for an unnamed bitfield. Such a bitfield forces alignment to the next storage unit boundary (which is a byte in NEURON C).

### ***Bitfield type is incorrect***

A bitfield may only be declared using only a combination of the type keywords int, signed, unsigned, long, short, and char. Bitfields may not be arrays, pointers, structures, or any other NEURON C type.

### ***Buffer size too small for interoperability***

The compiler issues warnings when any of the buffer size pragmas are used, and the resulting settings would prohibit satisfying the interoperability criteria.

### ***Buffer size too small for network management messages***

The compiler issues warnings when any of the buffer size pragmas are used, and the resulting settings would be too small to accommodate all possible network management messages from being properly received or acknowledged.

---

## **C**

### ***Call only applies to bindable msg\_tag***

The `is_bound()` built-in function returns TRUE (nonzero) if the requested object has been bound. Otherwise, it returns FALSE. The function applies only to network variables and to bindable message tags. A bindable message tag is a message tag declared *without* the `bind_info` (nonbind) option.

### ***Call to 'io\_out' requires output value parameter***

The built-in function call `io_out()`, which is used to initiate an output to a NEURON I/O object, must be given at least two parameters, the first being the I/O object name, and the second being the value to be output.

### ***Call to function without prototype***

NEURON C is more restrictive than ANSI C in this area. The NEURON CHIPS stack machine architecture does not permit calling undeclared functions with unknown numbers of parameters.

### ***Can't compute 'sizeof' for object***

The compiler attempted to calculate the size of a type, but did not have enough information. This could result from a `sizeof` expression for an object like a timer object, or a message tag, or a `typedef` name which is a `bind_info`, or some similar circumstance.

### ***Can't convert address of const into non-const ptr***

To prevent data that is declared constant from being modified, NEURON C will not permit pointers including the `const` attribute from being cast such that the `const` attribute

is removed. Neither does the compiler permit an implicit conversion of pointer (via function call, etc.) such that the `const` attribute would be removed.

***Can't enable `micro_interface` with `Net Vars` or `msg_tags` declared***

The `#pragma micro_interface` can only appear in a program if there have not been any prior declarations of network variables or message tags.

***Can't have a 'select' pin on 'neurowire slave' object***

The Neurowire master I/O object declaration requires a 'select' value. The Neurowire slave I/O object declaration does not.

***Can't have a 'timeout' pin on 'neurowire master' object***

The Neurowire slave I/O object declaration permits a 'timeout' value. The Neurowire master I/O object declaration does not.

***Can't have duplicate case labels with same value***

A switch statement cannot have ambiguous labels. It can have no more than one default label, and no two case labels may have the same value.

***Can't have 'io\_changes' & 'io\_update\_occurs' on same I/O object***

The NEURON C event expressions for `io_update_occurs` and `io_changes` (with its various options) only apply to I/O objects that are inputs. Furthermore, some events are not applicable to some input object types. Only one form of event expression can be used per I/O object. A maximum of 15 I/O objects can have `io_update_occurs` and `io_changes` events. The syntax of the event expressions all require the event type to be followed by the object name, in parentheses. For more information, see Appendix C.

***Can't have more than 14 bindable `msg_tags` & NVs too***

A NEURON CHIP has up to 15 outgoing message ports. (Ports are also known as "address table entries.") Each bindable

message tag consumes one port, whether bound or not. Network variables can share ports, but there must be at least one port available.

However, if there aren't enough entries available for each output network variable to have its own port, you get this warning. This is because the binder would then have to *share* the remaining address table entries among the network variables.

**EXAMPLE:** If there are three network variables (each going to a different destination) and there are only two address table entries, then at least two of the network variables would have to use the same address table entry (if they are all connected). Now let's assume that all the variables are connected, each point-to-point to a different node. If each variable had its own address table entry, the LONTALK messages would all use subnet/node (i.e. point-to-point) addressing.

However, for the two variables sharing the *same* entry, a group will be constructed. This means that, when either variable is updated, the updates will go to all members in the group. This does not cause a problem, as the nodes which don't have that variable throw the update away. The major inefficiency the compiler is warning about, though, is that each destination in the group, regardless of whether it uses the message, will respond with an acknowledgment message. This situation thus leads to unnecessary acknowledgements, or other extra network traffic.

***Can't have more than one default label per switch statement.***

A switch statement cannot have ambiguous labels. It can have no more than one default label, and no two case labels may have the same value.



***Can't modify a constant object***

***Can't modify via pointer-to-constant-object***

To prevent data that is declared constant from being modified, NEURON C will not permit constant objects to appear on the left-hand side of an assignment statement, nor will it permit modification of the constant object via a pointer with the `const` attribute, or via the `++` or `--` operators.

Note that, in the case of network variables, a network variable declared as `const` (or `config`, which implies `const`) cannot be modified in the node where it is so declared, but it *can* be modified by other nodes in the network.

***Can't open assembly output file***

***Can't open binder interface file***

***Can't open debug output info file***

***Can't open output dependency-file***

The compiler opens several output files as part of its initialization. All the files above are placed in the project directory in the subdirectory IM (which contains intermediate files from the build process). This error may indicate that the disk is full, or may indicate a disk error, or may indicate a read-only disk condition (if for example, the project directory and its subdirectories are on a floppy disk and the disk is write-protected).

Also, confirm that the project directory contains a subdirectory named IM.

***Can't open include file 'filename'***

***Can't open source file 'filename'***

The file which is named cannot be found. Check the spelling of the filename and check the Application Directories' and Include Directories' search paths in the LONBUILDER IDE Options/Project configuration screen.



### ***Can't open bplate.ns***

During compiler initialization, the compiler attempts to open several support files. One of these files is named `bplate.ns`. This file should reside in the LONBUILDER system directory (default `\lb`). This message could indicate a disk error.

### ***Can't re-declare 'bind\_info'***

The `bind_info` can appear at most once in the declaration of a network variable or a message tag. The `bind_info` cannot be combined with other `bind_info` by concatenation.

### ***Can't redefine 'name'***

The name has been declared as a certain type of identifier more than once in the scope. For example, it is illegal to define a variable twice at file scope, or a function, or a macro, etc. (however, there may be multiple extern declarations for the same variable). Note that a macro definition is always considered to be at file scope, regardless of its placement in a file.

### ***Can't remove 'const' attribute via cast operation***

To prevent data that is declared constant from being modified, NEURON C will not permit pointers including the `const` attribute from being cast such that the `const` attribute is removed. Neither does the compiler permit an implicit conversion of pointer (via function call, etc.) such that the `const` attribute would be removed.

### ***Can't repeat this pragma***

Some pragmas can only be used once. These are:

```
app_buf_out_size
app_buf_in_size
app_buf_out_priority_count
app_buf_out_count
app_buf_in_count
disable_snvt_si
enable_sd_nv_names
net_buf_out_size
net_buf_in_size
net_buf_out_priority_count
net_buf_out_count
net_buf_in_count
num_addr_table_entries
num_domain_entries
one_domain
receive_trans_count
set_id_string
set_netvar_count
set_node_sd_string
set_std_prog_id
```

### ***Can't take address of EEPROM object***

NEURON C does not permit taking the address of a variable in EEPROM.

### ***Can't take address of message object***

The message objects, `msg_out`, `msg_in`, `resp_out`, and `resp_in`, have no value in themselves. They only have meaning when they are accessed using the dot operator (.) and a field name. Only certain predefined field names apply. The data field is an array, and access to it must be via index, except when used with `memcpy()`.

### ***Case value is out of range***

Valid range is -128 to +127.

***Character constant is too long***

NEURON C only supports single-character constants (this does not apply to use of the \ escape character sequence).

***Character in input is not acceptable for C source***

The NEURON C compiler uses only the minimum ANSI C standard character set. Additionally, the characters \$, @, and ` (accent-grave) can be used in string and character constants. All other non-standard characters are treated as white space, except for ^D and ^Z. Appearance of either of these two characters in the input file is taken to be an end-of-file marker.

***Character sequence '..' is not a suitable C token***

This character sequence has no meaning in C. The optional parameter specifier for functions is "...", not "..".

***Class 'register' is ignored***

This keyword has no effect in NEURON C.

***Class 'config' applies only to 'input' variables***

***Class 'config' can only be used with network variables***

Network variable declarations can be modified with the keywords config and polled. The config keyword only applies to input variables. The polled keyword only applies to output variables.

***Class 'volatile' is ignored***

This keyword has no effect in NEURON C.

***Codegen buffer is full***

The compiler memory limitation has been exceeded and the procedure must be split into two or more procedures.

### ***Comment is possibly unterminated***

This warning may be useful in discovering unintentional comments in your NEURON C program. The definition of C does not permit nesting of comments. Any text of the form shown below is treated as a single comment.

```
/* <text> /* <text> */
```

Note, however, that this particular pattern may indicate a condition where there are actually two comments intended, but the first is unterminated. Thus, the compiler detects this condition and prints a warning message. The comment text is printed also, for up to 256 characters.

### ***Comment not properly terminated***

An end-of-file condition was discovered in the middle of a comment. This is an unterminated comment condition and is an error. The error message contains the beginning of the comment.

### ***Const variables require initialization***

A variable declared with the `const` attribute must have an initializer. Note that this does *not* apply to a typedef that includes the `const` attribute.

---

## ***D, E, F***

### ***Duplicate I/O object modifier not allowed***

Several of the I/O object declarations permit or require modifiers, like `mux`, `ded`, `sync`, and so on. These are permitted or required on a per I/O object-type basis. At most one of each type of modifier is permitted in a single declaration.

### ***Enum constant out of range***

#### ***Enum list has more values than the debugger supports***

The range of enum values in NEURON C is from -128 to 127. According to the definition of ANSI C, multiple enumerated constant names may appear in an enum type for the same constant value; thus there is really no limit to the number of names in an enum value list.

However, the NEURON C Debugger only supports a maximum of 255 enumerated constant names in a given enum type. An enum that contains more names than this is still perfectly acceptable to the compiler; however, the debugger is only capable of using the first 255 names in the enum type.

#### ***Enum value wrapped around to negative***

The enumerated type automatically assigns consecutive integers as the values of the named constants unless specifically given a value for a constant. When `<constant n>` has the value 127, and `<constant n+1>` appears, the compiler automatically assigns it the "next" value. In NEURON C, this value is -128 (since enums are signed by definition in ANSI C). The compiler issues a warning diagnostic for this condition, and proceeds.

#### ***Event 'nv\_update\_occurs' only applies to input variables***

The event `nv_update_occurs` accepts one optional parameter which must be the name of a previously declared input network variable.

#### ***Explicit addressing requires inclusion of <MSG\_ADDR.H>***

An attempt to use the `msg_out.dest_addr` field or the `msg_in.addr` field has been detected, but cannot be compiled because the include file `<MSG_ADDR.H>` was not included by the programmer. Note that the include directive must appear *prior* to the first such field reference. The include file is *not* needed for references to other fields of the `msg_out` or `msg_in` objects.



### ***Expression for switch must be a 'short'***

The switch statement can handle values only in the range of the int data type, namely from -128 to 127 inclusive.

### ***Expression has no effect - discarded***

The compiler outputs this warning diagnostic when an expression is discarded by the optimizer. Examples of such expressions are:

```
x = y-1, z;          /* y-1 is discarded */
a+3;                 /* a+3 is discarded */
x == 1? y, z;        /* z is discarded */
```

### ***Expression must evaluate to a constant***

Expressions in certain NEURON C declarations and initialization statements must evaluate to compile-time integer constants.

### ***Expression type mismatch***

These error diagnostics result from combining expressions of conflicting types, such as assigning an int to a pointer, or a pointer of one type to a pointer to another type, or in using objects that have no value (such as message tags or I/O object names) in expressions.

In many cases in ANSI C, you must use an explicit type cast. However, note that casting should be avoided if possible, as it is often poor programming practice.

### ***Extern declarations cannot have initializers***

The semantics of an extern declaration and an initialized declaration are incompatible. An extern declaration is intended to reference an object defined elsewhere (usually in another module, although it may be a forward reference). An initialized declaration is intended to be the defining declaration of the object. Only one such initialized declaration should appear for each object.



***Extra entries in preprocessor directive***

This error indicates that, although the preprocessor directive was of the correct syntax, there are additional entries on the line that were not part of the directive.

***Event conflict for I/O object 'name'***

A single I/O object cannot be used in both an `io_update_occurs` event *and* an `io_changes` event.

***Field name in struct/union cannot be repeated***

Each field name at a given level of a `struct` or `union` declaration must be unique. Names are case sensitive.

***File write error - is disk full?***

The compiler encountered a DOS error writing to the output file(s). Check that the output media is not write-protected, and that sufficient disk space exists. It is possible, for extremely large programs, that a megabyte or more of temporary disk space would be needed during compilation.

***Float constants are not supported******Floating point is not supported***

NEURON C does not support any aspect of floating point numbers.

***Function declarations must use prototypes***

NEURON C is more restrictive than ANSI C in this area. The NEURON CHIPS stack machine architecture does not permit calling undeclared functions with unknown numbers of parameters.

***Function definition does not allow return value***

The function, whose declared return data type is `void`, has a statement of the form `return <expression>`.

***Function does not allow parameters***

The compiler outputs these diagnostics when the number of actual parameters or the actual parameter types, do not match those in the prototype, and they cannot be automatically converted.

***Function must return a value***

The function, whose declared return data type is *not* void, does not have a return statement (in every possible path to the end of the function) that returns a value of the appropriate type.

***Function parameter may not be 'struct' or 'union' type***

NEURON C does not support passing struct or union types by value as function parameters. You may use a pointer to the struct or union as a function parameter.

---

***H,I,K******Hex escape char code constant is too large***

A hex escape character inside a character or string constant exceeds the value 0xFF. The NEURON C behavior is correct for ANSI C, but programmers usually find the ANSI C behavior in this respect counter-intuitive.

***I/O events only apply to input objects***

The NEURON C event expressions for `io_update_occurs` and `io_changes` (with its various options) only apply to I/O objects that are inputs. Furthermore, some events are not applicable to some input object types. Only one form of event expression can be used per I/O object. A maximum of 15 I/O objects can have `io_update_occurs` and `io_changes` events. The syntax of the event expressions all require the event type to be followed by the object name, in parentheses. For more information, see Appendix C.

### ***I/O function call requires arguments***

Insufficient arguments (or no arguments) were passed to the I/O built-in call flagged by the compiler diagnostic. All I/O functions require at least one argument, namely the I/O object name.

### ***I/O function not valid for this I/O object***

Some built-in functions, such as `io_set_clock()` and `io_select()`, cannot be used on all I/O object types.

### ***I/O object requires 'master' or 'slave'***

The Neurowire I/O object being declared must have either 'master' or 'slave' after the keyword 'neurowire'.

### ***I/O object requires 'master', 'slave', or 'slave\_b'***

The parallel I/O object type declaration must be qualified with one of the keywords `master`, `slave`, or `slave_b` to specify which parallel I/O protocol is to be used.

### ***I/O object requires 'select' pin specification***

The Neurowire I/O object type declaration must also include specification of a pin to be used for an I/O object select. Only a pin from `IO_0` through `IO_7` may be used for a select pin.

### ***I/O object requires 'sync' pin on IO\_4***

### ***I/O object requires 'sync' pin on one of IO\_4...IO\_7***

### ***I/O object requires 'sync' pin specification***

The triac I/O object type declaration must include an assignment for a sync pin. Since the triac type uses a timer/counter output, the sync pin must use a corresponding input on the same timer/counter. If the dedicated timer/counter is used, only `IO_4` can be used for the sync pin. If the multiplexed timer/counter is used, any of `IO_4`, `IO_5`, `IO_6`, or `IO_7` can be used.

***I/O object type cannot have an initial-pin-level***

Most output object types permit specification of an initial pin-level value to be assumed by the pin on power up or chip reset, until the application program takes over. All single-pin initial levels must be either 0 or 1. The nibble I/O object type, which uses four consecutive pins, can have initial values from 0 to 15, with the values being mapped as a binary number onto the four pins. Likewise, the byte I/O object type can have initial values from 0 to 255.

***I/O object type not allowed on pin IO\_7 or IO\_10***

***I/O object type not available on requested pin***

***I/O object type restricted to pin IO\_0***

***I/O object type restricted to pins IO\_0 through IO\_4***

***I/O object type restricted to pin IO\_8***

***I/O object type restricted to pin IO\_10***

***I/O object type restricted to pins IO\_0 or IO\_1***

***I/O object type restricted to pins IO\_0 through IO\_7***

***I/O object type restricted to pins IO\_4 or IO\_6***

***I/O object type restricted to pins IO\_4 through IO\_7***

Different I/O object types are permitted on different subsets of the NEURON CHIP's I/O pins. For more information, see Appendix C.

***I/O objects can only be declared at file scope***

Some NEURON C objects may only be declared at file scope. Thus, they are restricted to being globals, not automatics. These objects are timer objects, message tags, I/O objects, and network variables.

***If one priority count is zero, both must be zero***

The programmer is allowed to control the counts and sizes of various buffers used in sending and receiving messages and network variable updates.

There are two priority buffer pools, one at the network level, and one at the application level. Either both pools must be empty, or both pools must contain buffers. A zero count cannot be specified for one priority pool and a nonzero count for the other.

***Implicit pointer conversion is not permitted***

ANSI C does not permit a pointer of one type to be implicitly converted to a pointer of another type by assignment or by passing as a function parameter. Use explicit casting.

***Improper binary constant 'text'***

A binary constant begins with 0b and is followed by one or more binary digits, (i.e. the digits 0 or 1).

***Improper context for 'break' statement***

***Improper context for case label***

***Improper context for 'continue' statement***

***Improper context for default label***

Certain statements in ANSI C do not have meaning except within some defined construct. The continue statement can only be used inside a loop statement, which is either a for, a while, or a do-while. The break statement can be used inside a loop statement or inside a switch statement.

The words case and default are reserved words in ANSI C, used as labels inside of the scope of a switch statement. They cannot be used outside of a switch statement.



### ***Improper initializer format***

A set of initializers in braces has too many levels of braces, or is otherwise incorrectly formulated. Initializers of aggregates (arrays, struct 's, or union 's) should have a set of braces for each level of aggregate, but individual values should not have their own braces.

### ***Improper octal constant 'number'***

An octal constant begins with 0, and is followed by one or more octal digits, (i.e. the digits 0-7).

### ***Incomplete binary constant 'text'***

A binary constant begins with 0b and is followed by one or more binary digits, (i.e. the digits 0 or 1).

### ***Incomplete hex constant 'text'***

A hexadecimal constant begins with 0x and is followed by one or more hexadecimal digits, (i.e. the digits 0-9, and the letters a-f and A-F).

### ***Incorrect 'clock' select value***

For I/O objects that accept a clock modifier in their declaration, the legal values are from 0 to 7, inclusive, except for the pulsecount output object, which uses only 1 to 7. The clock value must be a constant expression.

### ***Incorrect 'numbits' value or type***

The bitshift I/O object type declaration can optionally specify the number of bits to be specified. This numbits modifier has as an argument that must be a compile-time integer constant expression with a value from 1 to 128.



### ***Incorrect I/O object type for changes-by event***

### ***Incorrect I/O object type for changes-to event***

### ***Incorrect I/O object type for io\_changes event***

### ***Incorrect I/O object type for I/O update-occurs event***

The NEURON C event expressions for `io_update_occurs` and `io_changes` (with its various options) only apply to I/O objects that are inputs. Furthermore, some events are not applicable to some input object types. Only one form of event expression can be used per I/O object. A maximum of 15 I/O objects can have `io_update_occurs` and `io_changes` events. The syntax of the event expressions all require the event type to be followed by the object name, in parentheses. For more information, see Appendix C.

### ***Incorrect message object field reference***

The message objects, `msg_out`, `msg_in`, `resp_out`, and `resp_in`, have no value in themselves. They only have meaning when they are accessed using the dot operator (`.`) and a field name. Only certain predefined field names apply. The data field is an array, and access to it must be via index, except when used with `memcpy()`.

### ***Incorrect number of parameters***

The compiler outputs these diagnostics when the number of actual parameters or the actual parameter types, do not match those in the function prototype, and they cannot be automatically converted.

### ***Incorrect use of 'void' in function prototype***

The `void` type has no size. It cannot be used as an argument of the `sizeof` operator, nor can it be used to declare a variable. Its only legal uses are in declaring function return types, declaring that a function has no parameters, and in combination with `*` to define a type `'void *'` (a wildcard pointer type).

***Initial-pin level must be 0 or 1***

***Initial-pin level must be in range 0...15***

***Initial-pin level must be in range 0...255***

Most output object types permit specification of an initial pin-level value to be assumed by the pin on power up or chip reset, until the application program takes over. All single-pin initial levels must be either 0 or 1. The nibble I/O object type, which uses four consecutive pins, can have initial values from 0 to 15, with the values being mapped as a binary number onto the four pins. Likewise, the byte I/O object type can have initial values from 0 to 255.

***Input network variables cannot have service-type***

Some of the options in the `bind_info` declaration modifier only apply to any network variable, some only apply to an output network variable, and some only apply to a message tag.

***Integer constant 'number' is too large***

Integer constants are limited to 65535, since the maximum size of an integer is 16 bits.

***Invalid 2nd argument to 'sleep'***

The built-in function `sleep()` has an optional second parameter which must be a previously declared NEURON C I/O object name. This I/O object is the one whose primary pin will be monitored for a wakeup condition. This pin must be one of `IO_4`, `IO_5`, `IO_6`, or `IO_7`.

***Invalid array bounds***

In NEURON C, an array bound constant expression must resolve to a positive integer no larger than 32767.

***Invalid cast operation***

Some conversions between data types are not permitted, even via an explicit cast. For example, an object cannot be cast if its base type is not known. Nor can an object be cast to `void` and then used in an expression.

### ***Invalid data type combination***

This diagnostic message results from incorrect or conflicting type combinations. For example, `short` and `long` is a conflicting type combination. Combining timer objects and message tags, for example, is an incorrect result type.

### ***Invalid indirection expression - not a pointer***

The operand of the `*` indirection operator is not a pointer. This operator can only be applied to a pointer variable or a constant typed as a pointer.

### ***Invalid operand for address operator***

The operand of the `&` address operator is not a variable, or is a variable type for which addressing is not permitted. For example, you cannot take the address of a numeric constant. In NEURON C, you also cannot take the address of a timer object, a message tag, a network variable, or an I/O object.

### ***Invalid operation on pointer***

The only binary operations permitted on pointers are `+`, `-`, and comparisons. A pointer can be added to a constant (or vice versa), and ANSI C scaling rules apply to the constant. Likewise, a constant can be subtracted from a pointer (but *not* vice versa). Finally, two pointers of the same type can be subtracted, one from the other. The result is a difference scaled by the size of the object type pointed to.

Pointers cannot be used in unary expressions other than with increment and decrement operators.

### ***Invalid parameter declaration in function***

This diagnostic results from certain errors in function definition syntax such as in the example below:

```
void f(int, long) { <fn body> }
```

### ***Invalid preprocessor directive syntax***

This error indicates one of any number of syntax problems in the # directive of the line indicated. The proper syntax is:

```
#<directive> [<value>]
```

where the optional <value> is dependent on the particular directive.

### ***Invalid storage class combination***

This diagnostic results from incorrect or conflicting combinations of storage class keywords, such as `eprom` and `ram`.

### ***Invalid struct/union field declaration***

#### ***Invalid struct/union field type***

A field in a `struct` or `union` may not have a storage class, and may not contain the word `typedef`. Nor can it be a message tag, a timer object, nor a network variable, nor can it have `bind_info`. Note also that a `union` may not contain bitfields.

#### ***Invalid subscript operation***

The compiler outputs this diagnostic when an array-index operator [ <expr> ] is applied to a variable that is not a pointer or an array.

#### ***Invalid type for array index***

The array index of the subscript operator must be an `int` or `char` type.

It may be `short` or `long`, signed or unsigned.

#### ***Invalid type for bitfield***

A bitfield may only be declared using only a combination of the type keywords `int`, `signed`, `unsigned`, `long`, `short`, and `char`. Bitfields may not be arrays, pointers, structures, or any other NEURON C type.

### ***Invalid type for subscript operation***

The object being subscripted (the “array”) must be either an array or a pointer. The type of the subscript must be an integer type.

### ***Invalid typedef id ‘name’***

Reference to symbol ‘name’ seemed to be a reference to a typedef identifier, but no such typedef was declared.

### ***Invalid use of pointer in binary operation***

The only binary operations permitted on pointers are +, -, and comparisons. A pointer can be added to a constant (or vice versa), and ANSI C scaling rules apply to the constant. Likewise, a constant can be subtracted from a pointer (but *not* vice versa). Finally, two pointers of the same type can be subtracted, one from the other. The result is a difference scaled by the size of the object type pointed to.

Pointers cannot be used in unary expressions other than with increment and decrement operators.

### ***Invalid use of reserved word >> word <<***

This error message occurs when the token causing the syntax error is a reserved word (keyword) which is used out of context. Check the syntax of not only the reported token, but a few previous tokens. Don’t forget that NEURON C has some extra reserved words, in addition to those provided by ANSI C. Check that you haven’t accidentally used a reserved word as a variable name or other identifier. NEURON C reserved words are listed in Section C.6 in Appendix C.

### ***Invalid use of type***

The compiler attempted to calculate the size of a type, but did not have enough information. This could result from a sizeof expression for an object like a timer object, or a message tag, or a typedef name which is a bind\_info, or some similar circumstance.



### ***Invalid use of void type***

The void type has no size. It cannot be used as an argument of the sizeof operator, nor can it be used to declare a variable. Its only legal uses are in declaring function return types, declaring that a function has no parameters, and in combination with '\*' to define a type 'void \*' (a wildcard pointer type).

### ***Invalid value for this pragma***

The numeric value following the pragma is not of appropriate value. Consult the documentation for the specific pragma to ascertain the applicable valid values. Pragmas are documented in Chapters 1 and 6.

### ***Keyword 'polled' is ignored for input network variable***

Network variable declarations can be modified with the keywords config and polled. The config keyword only applies to input network variables. The polled keyword only applies to output network variables.

---

## **L, M**

### ***Label 'name' is not defined***

The specified label used in a goto statement is not defined in the current procedure or task.

### ***Line too long in macro definition***

No input line in NEURON C can exceed 256 characters. You can use the line continuation feature of ANSI C to extend the line. This feature is activated by using a \ character at the end of the line.

### ***Long constant value being converted to short***

These diagnostics result from an automatic conversion of a long variable to a short. To make these warnings go away, modify the variable declarations or use an explicit cast operator.



***Macro name being defined is too long***

No identifier in NEURON C can exceed 256 characters

***Macro text for 'macro name' is too long for debugger***

The debugger can understand macro names, but can only handle the first 16K of text used in the definition of a macro.

***Macros with arguments 'macro name' aren't supported***

NEURON C does not support definition of macros with arguments.

***Maximum token length exceeded***

No NEURON C token can exceed 256 characters. This applies to identifiers, numbers, string constants, and so on. This does *not* apply to comments.

***Message event code must be in range 0-127***

The event `msg_arrives` accepts one optional parameter, which is a message event code. This code must be a compile-time constant integer expression which has a value from 0 to 127, inclusive.

***Message object reference cannot be assigned to***

The message objects `msg_in` and `resp_in` are read-only. Attempts to use these objects on the left side of an assignment statement result in the diagnostic message above.

***Message object reference has no value***

The message objects, `msg_out`, `msg_in`, `resp_out`, and `resp_in`, have no value in themselves. They only have meaning when they are accessed using the dot operator (.) and a field name. Only certain predefined field names apply. The data field is an array, and access to it must be via index, except when used with `memcpy()`.

### ***Mixing fn prototypes and old-style parm list not allowed***

This diagnostic results from a declaration of the form:

```
void f(int a, b) int b; { <fn body> }
```

Use only new-style ANSI C function declaration syntax:

```
void f(int a, int b) { <fn body> }
```

or old-style (traditional) C function declaration syntax:

```
void f(a,b) int a; int b; { <fn body> }
```

Do not intermix these two styles within a single function definition.

### ***Must specify 'enable\_multiple\_baud' for correct I/O operation***

The `#pragma enable_multiple_baud` directive must appear *prior* to the use of any I/O function (e.g. `io_in()`, `io_out()`). If this error message appears, move the `#pragma enable_multiple_baud` directive to the beginning of your program.

### ***Must specify 'input' or 'output' for this I/O object type***

Some NEURON C I/O object types can only be “input”, some can only be “output,” and some can be either. For the case where the direction is known from the I/O object type, the programmer need not specify the direction, but if specified, it must be the correct direction. For the case where the direction is not known from the I/O object type, the programmer must specify it.

---

## **N**

### ***Name of msg\_tag, 'name', exceeds 16 chars***

### ***Name of network variable, 'name', exceeds 16 chars***

Any network variable or message tag name is limited to 16 characters, as this is the maximum length of identifiers in the object database. Likewise, if a `typedef` name is used in declaration of a network variable, it too must be 16 characters or less in length.

### ***'name' is a restricted symbol***

The 'name' shown cannot be used for a user-declared identifier, macro, etc. All such names begin with '\_', although not all names beginning with the '\_' character are reserved. To avoid this problem, don't declare any names beginning with the '\_' character.

### ***'name' is not defined***

An identifier was used in an expression that was not previously declared or defined. ANSI C requires that all identifiers be declared before their first use.

### ***Need to check msg\_fails for tag 'tagname'***

### ***Need to check msg\_succeeds for tag 'tagname'***

### ***Need to check nv\_update\_fails for 'nv\_name'***

### ***Need to check nv\_update\_succeeds for 'nv\_name'***

Some events must occur in pairs. For a given network variable or message tag, checking one event requires checking of a corresponding event, to have a correct program.

### ***Name is not an I/O object name***

The first argument passed to the flagged I/O built-in call is not a properly declared I/O object name. Note that in ANSI C, a general rule is that an object must be declared before its first use.

### ***Net variable base type can't contain function***

### ***Net variable base type can't contain pointer***

### ***Network variable array bound is incorrect***

This message indicates that the array bound portion of a network variable declaration, or a network variable event expression, is not in the valid range, or not of the proper format (e.g. an attempt to be multiply dimensioned, or an index of < 1).

### ***Network variable base type can't contain unbounded array***

Network variable arrays must be declared with a fixed bound that is a compile-time constant.

***Network variable declaration not permitted if  
micro\_interface***

Once the `#pragma micro_interface` appears, the program cannot declare any network variables or message tags.

***Network variables cannot be declared as non-bindable***

Some of the options in the `bind_info` declaration modifier only apply to any network variable, some only apply to an output network variable, and some only apply to a message tag.

***Neurowire slave device can't have a baud or kbaud specifier***

The Neurowire master device generates the clock used in the transfer. Therefore, specification of a bit rate in the declaration of a slave Neurowire device is meaningless.

***No declaration for formal parameter - int assumed***

The definition of ANSI C permits a declaration at file scope without a type. Likewise, functions may be declared without a return type. Also it is possible to construct a typecast which does not actually contain a type. Such declarations must default to `int`, by the ANSI C definition. However, such declarations are poor programming practice, and may even indicate an error, thus the compiler issues a warning diagnostic.

Consider the following example:

```
unsigned long x1, x2; x3;
```

Note the semicolon following `x2`. This is most likely a typo, however, ANSI C permits this and results in `x3` being declared by default as an `int`. Due to white space rules, this appears the same to the compiler as the following declaration:

```
unsigned long x1, x2;  
x3;
```

This is almost certainly *not* what the programmer intended, yet most C compilers do not issue a warning in these circumstances.

### ***No formal parameter matches the parameter declaration***

This diagnostic results from an error of the form shown below, where there is no declaration for the parameter named b.

```
void f(a,b) int a; { <fn body> }
```

### ***Not a field in specified struct/union***

The compiler outputs these diagnostics when the right-hand side of the '-' or '.' operator is not a field in the struct or union type that corresponds to the left-hand-side expression.

### ***Not enough address table entries for optimum efficiency***

#### ***Not enough address table entries***

A NEURON CHIP has up to 15 outgoing message ports. (Ports are also known as “address table entries.”) Each bindable message tag consumes one port, whether bound or not. Network variables can share ports, but there must be at least one port available.

However, if there aren't enough entries available for each output network variable to have its own port, you get this warning. This is because the binder would then have to *share* the remaining address table entries among the network variables.

**EXAMPLE:** If there are three network variables (each going to a different destination) and there are only two address table entries, then at least two of the network variables would have to use the same address table entry (if they are all connected). Now let's assume that all the variables are connected, each point-to-point to a different node. If each variable had its own address table entry, the LONTALK messages would all use subnet/node (i.e. point-to-point) addressing.



However, for the two variables sharing the *same* entry, a group will be constructed. This means that, when either variable is updated, the updates will go to all members in the group. This does not cause a problem, as the nodes which don't have that variable throw the update away. The major inefficiency the compiler is warning about, though, is that each destination in the group, regardless of whether it uses the message, will respond with an acknowledgment message. This situation thus leads to increased unnecessary acknowledgements, or other extra network traffic.

#### ***Not enough parameters passed to function***

The compiler outputs these diagnostics when the number of actual parameters or the actual parameter types, do not match those in the prototype, and they cannot be automatically converted.

#### ***Nothing was declared***

Similar to *Declaration defaults to 'int'* No declaration for formal parameter - int assumed, in a situation like the following:

```
long; int j;
```

This is legal C, but may not be what the programmer intended. Thus, for the "first" declaration (the statement "long;") this diagnostic is produced. Many C compilers do not issue a warning in these circumstances. Note that NEURON C will *not* issue a warning in the following cases, since something *is* being declared (namely the enum's or the struct field names, respectively):

```
enum { FALSE, TRUE } ;  
struct { int a; int b; };
```



***num\_addr\_table\_entries adjusted upward to 'number'***

If msg\_tags are declared, or network variables are used, the compiler computes a minimum allowable number of address table entries. There must be one entry per bindable msg\_tag declared, plus at least one entry if network variables are used. For some possible connections, this may not be enough entries. However, this is an absolute minimum.

---

**O, P**

***Object being declared cannot be initialized***

Declaration-time initialization cannot be used for typedefs, timer objects, message tags, function parameters, structure tags, union tags, and enum tags.

***Object cannot be a function parameter***

Objects that have no type (such as message tags or I/O objects) cannot be function parameters. Likewise, if p were declared void \*, \*p would not be a valid function parameter (or any other expression, for that matter).

***Object is not a struct/union pointer***

***Object is not a structure or union***

The compiler outputs these diagnostics when the left-hand side of the -> or . operator, respectively, is not a pointer to a struct or union type.

***Object is not a suitable assignment target***

The left-hand side of assignment operators, and the target of increment or decrement operators must be nonconstant variables, or fields of nonconstant structures or unions, or elements of arrays.

### ***Object of call is not a function***

The syntax encountered is function call syntax, i.e.:

```
<expression> ( [<expression list>] )
```

however, the <expression> being called is not a function (or a pointer to a function). Note that this error could occur by omitting an operator.

If the following were intended:

```
int a, b, c, d;  
a = b * (c + d);
```

but the following were actually written (omitting the multiplication operator):

```
int a, b, c, d;  
a = b (c + d);
```

This would appear to be a function call, but b is not a function.

### ***Offline does not apply to a msg\_tag***

Some of the options in the `bind_info` declaration modifier only apply to network variables, some only apply to output network variables, and some only apply to message tags.

### ***Parameter must be a msg\_tag***

The events `msg_completes`, `msg_succeeds`, and `msg_fails` all accept one optional parameter which must have been previously declared as a message tag.

### ***Parameter must be a network variable***

The events `nv_update_completes`, `nv_update_fails`, and `nv_update_succeeds` all accept one optional parameter which must be the name of a previously declared network variable.

### ***Parameter must be a timer name***

The event `timer_expires` accepts one optional parameter which, if supplied, must be the name of a previously declared timer object.

***Parameter must be an I/O object name***

The NEURON C event expressions for `io_update_occurs` and `io_changes` (with its various options) only apply to I/O objects that are inputs. Furthermore, some events are not applicable to some input object types. Only one form of event expression can be used per I/O object. A maximum of 15 I/O objects can have `io_update_occurs` and `io_changes` events. The syntax of the event expressions all require the event type to be followed by the object name, in parentheses. For more information, see Appendix C.

***Parameter must be either a msg\_tag name or an NV name***

The `is_bound()`, `addr_table_index()`, and `nv_table_index()` built-in functions return TRUE (nonzero) if the requested object is bound (connected). Otherwise, they return FALSE. The functions apply only to network variables and to bindable message tags. A bindable message tag is a message tag declared *without* the `bind_info` (nonbind) option.

***Parameter to 'poll' must be input network variable***

The built-in function `poll()` takes as its only argument the name of an input network variable.

***Pin IO\_0 requires use of 'mux' timer/counter***

***Pin IO\_1 requires use of 'ded' (dedicated) timer***

***Pin IO\_4 needs 'mux' or 'ded' specification***

***Pins IO\_5...IO\_7 must use 'mux' timer***

For I/O object types that use a timer/counter, the timer/counter used is dependent on the pin assigned to the I/O object. There are two timer/counters, the dedicated (abbreviated `ded`) and the multiplexed (abbreviated `mux`). The dedicated circuit uses pin `IO_1` for output and pin `IO_4` for input. The multiplexed circuit uses pin `IO_0` for output and multiplexes among pins `IO_4`, `IO_5`, `IO_6`, and `IO_7` for input.

The programmer need not specify which timer/counter circuit is being used except when the I/O object is assigned to

pin IO\_4. Then, either the mux or ded keyword must be included in the declaration of the I/O object.

#### ***Pin/resource conflict with a previous I/O object***

In several cases, more than one NEURON CHIP I/O object type can be assigned to a single pin within a single application. The rules for overlaying I/O object declarations are discussed in Chapter 2 of this guide.

#### ***Possible data loss converting long to short***

##### ***Possible data truncation***

These diagnostics result from an automatic conversion of a long variable to a short. To make these warnings go away, modify the variable declarations or use an explicit cast operator.

#### ***Pragmas 'hidden' and 'no\_hidden' only allowed in 'echelon.h'***

The pragmas #pragma hidden and #pragma no\_hidden are for internal use from the standard include file echelon.h only. Do not use them.

#### ***Preprocessor # directives must begin in column 1***

Although ANSI C specifies that the # of the preprocessor directive need only be the first non-blank character on the line, the NEURON C Compiler requires that it be the first character on the line, with no preceding spaces.

#### ***Preprocessor directives cannot be nested in macros***

A macro cannot contain the character # outside of the text of a string or a character constant.

***Rate estimate value is out of valid range***

The `bind_info` permits specification of average and maximum message rate estimates for each tag or network variable. The valid range of rate estimate values is from 0 to 18780, in units of TENTHS of a message per second. Thus, a specified value of 1043 indicates an actual value of 104.3 messages per second.

***Read error on file 'file name'***

A DOS read error was reported while reading the file of the given name.

***Recommend use of 'scheduler\_reset' feature***

The compiler makes this recommendation when there is a possibility of anomalous execution of different tasks because their respective *when* clauses are not mutually exclusive.

***Recursive include file nesting 'file name' not allowed***

An include file cannot include itself, nor can any file B included from file A then re-include file A, etc.

***Recursion of macro 'macro name' is not allowed***

A macro cannot expand such that the expansion includes the name of a macro. This is because, after macro expansion, the resulting text is re-scanned for additional macro substitution.

***Redefinition of 'name' hides enum value******Redefinition of 'name' hides function******Redefinition of 'name' hides label***

The macro being defined, by the rules of ANSI C, will supersede any other declarations of the same name. This may not be what the programmer intended, so if a conflict is detected, the above warning(s) will be printed.



### ***Repeated bind\_info option***

The `bind_info` keyword is followed by a parenthesized list of one or more options, some with associated values. Each keyword may appear at most once.

### ***Repeated data type keyword ignored***

Repeated keywords are ignored (for example, “`const const`” is the same as “`const`”, “`int int`” is the same as “`int`”), but a diagnostic message is printed.

### ***Repeated keyword is ignored***

The keyword `const` or `volatile` is used more than once in modification of a pointer type.

### ***Repeated storage class keyword ignored***

Repeated keywords are ignored (for example, “`const const`” is the same as “`const`”, “`int int`” is the same as “`int`”), but a diagnostic message is printed.

### ***Return value of function ignored***

A function that has a non-void return type is used in an expression, but the return value is discarded without being used or stored. The warning can be removed by casting the return of the function to void.

Example:

```
int f(void) {return 0;}
when (reset) {
    (void)f();
}
```

### ***SD string supplied exceeds 1023 character limit***

The ‘`sd_string`’ option for a network variable declaration is limited to a string of no more than 1023 characters (plus NUL terminator).



### ***Service types may not be specified for a msg\_tag***

Some of the options in the `bind_info` declaration modifier only apply to any network variable, some only apply to an output network variable, and some only apply to a message tag.

### ***Sleep wakeup I/O object must be input on IO\_4..IO\_7***

The built-in function `sleep()` has an optional second parameter which must be a previously declared NEURON C I/O object name. This I/O object is the one whose primary pin will be monitored for a wakeup condition. This pin must be one of `IO_4`, `IO_5`, `IO_6`, or `IO_7`.

### ***Special event & init code block exceeds size limitation***

The tasks corresponding to the reset, online, offline, and wink events, as well as any when clause arbitrary expressions all generate code in a special area known as the APINIT block. If the `#pragma disable_mult_module_init` directive is used, any non-zero initialization of global RAM variables and I/O objects place code here as well. This block is limited in size to 255 bytes.

If you exceed the size of this block, try moving the bulk of code in any tasks that correspond to reset, online, offline, and wink events to functions that are called from these tasks. If you are using the `#pragma disable_mult_module_init` directive, remove the pragma.

### ***Specify '#pragma enable\_multiple\_baud' for correct I/O operation***

Two or more I/O devices have been declared with conflicting baud rates. In order for the compiler to generate correct code, it must know about the conflicting devices in advance. Thus, the `#pragma enable_multiple_baud` directive must be specified in advance.

### ***Storage class on struct/union field not permitted***

A field in a struct or union may not have a storage class, and may not contain the word `typedef`. Nor can it be a message tag, a timer object, nor a network variable, nor can it have `bind_info`. Note also that a union may not contain bitfields.

### ***String constant is not terminated***

ANSI C does not permit a string constant to span lines. Nor can a string constant be terminated by end-of-file. To create a very long string constant, use the ANSI C string constant concatenation feature, demonstrated below. Note that the parts of the string are concatenated without insertion of any white space, newline, or other separator character.

```
"This is a long string constant "
```

```
"split across two source lines."
```

Note that this error message may also indicate mismatched quotes in strings.

Use `\` to include a quote character in a string constant.

### ***Syntax error when reading >>'token'<<***

Any syntax error is reported in this manner. Check the syntax of not only the reported token, but the last few previous tokens. Don't forget that NEURON C has some extra reserved words, in addition to those provided by ANSI C. Check that you haven't accidentally used a reserved word as a variable name or other identifier. Reserved words are listed in *Section C.7* in Appendix C.

### ***System open file limit exceeded***

DOS limits the number of files that may be open simultaneously, and this is reflected in the compiler. The number of files is limited by the `FILES=` setting in `CONFIG.SYS` (see your DOS manual). We recommend that a setting of at least 20 is used. However, some configurations (use of TSRs with files, deeply nested include file structures, and so on) may require a larger `FILES=` setting. Try increasing this value (you must reboot

to have any change to CONFIG.SYS take effect). However, under no circumstances is any one process allowed to have more than 20 files open simultaneously. Thus, the practical limit to nesting of include files is 12 deep (as the compiler may have as many as eight non-include files open at once).

---

## T, U

### ***The 'priority' is ignored for this 'when' clause***

There are three special events in NEURON C which can only appear in at most one *when* clause. These events are reset, offline, and online. The declaration of priority for these clauses has no effect because, due to the special times at which these clauses are executed, they always have priority.

### ***The quad type is not supported.***

The quad type is not supported by NEURON C, but quad is a reserved word.

### ***The 'select' pin must be one of IO\_0...IO\_7***

The Neurowire I/O object type declaration must also include specification of a pin to be used for an I/O object select. Only a pin from IO\_0 through IO\_7 may be used for a select pin.

### ***The 'timeout' pin must be one of IO\_0 ... IO\_7***

The Neurowire slave's 'timeout' pin option can only be one of the pins IO\_0 through IO\_7.

### ***This declaration may only be at file scope***

Some NEURON C objects may only be declared at file scope. Thus, they are restricted to being globals, not automatics. These objects are timer objects, message tags, I/O objects, and network variables.

### ***This event cannot be duplicated***

There are three special events in NEURON C which can only appear in at most one when clause. These events are reset, offline, and online. The declaration of priority for these clauses has no effect because, due to the special times at which these clauses are executed, they always have priority.

### ***This event duplicates or overlaps a previous one***

In many cases, use of a *when* clause containing an event that is a duplicate or an overlap of a previous event expression would prevent the associated task from being executed, or may cause anomalous behavior, with one task being executed sometimes, and the other being executed the rest of the time.

(This latter behavior would occur as the result of round-robin execution by the NEURON CHIP firmware scheduler, if the `scheduler_reset` feature were not used.)

### ***This event expr is not permitted - firmware restriction***

The special event keywords `offline`, `online`, and `wink` cannot be combined into other expressions when used in the *when* clause. See the *Additional Predefined Events* section in Chapter 5 for more explanation.

### ***This event will never be reached***

This message warns of the use of a specific, qualified event following a generic, unqualified event in the same class. As the generic one will catch the event first, the specific one will never evaluate to TRUE. This condition can only occur when using the `scheduler_reset` feature. Failure to use the `scheduler_reset` feature can result in unstable behavior.

### ***This field must be indexed***

The message objects, `msg_out`, `msg_in`, `resp_out`, and `resp_in`, have no value in themselves. They only have meaning when they are accessed using the dot operator (`.`) and a field name. Only certain predefined field names apply. The data field is an array, and access to it must be via index, except when used with `memcpy()`.

***This I/O object type can only be 'input'***

***This I/O object type can only be 'output'***

Some NEURON C I/O object types can only be "input", some can only be "output," and some can be either. For the case where the direction is known from the I/O object type, the programmer need not specify the direction, but if specified, it must be the correct direction. For the case where the direction is not known from the I/O object type, the programmer must specify it.

***Too many function parameters for debugger***

The NEURON C Debugger can only support functions with 14 or fewer parameters. Use of functions with more than 14 parameters may result in strange or incorrect results when using the debugger to display stack contents, and so on. Note that this is a limitation on number of parameters, and *not* on the number of bytes used to store those parameters.

***Too many I/O object change events used***

The NEURON C event expressions for `io_update_occurs` and `io_changes` (with its various options) only apply to I/O objects that are inputs. Furthermore, some events are not applicable to some input object types. Only one form of event expression can be used per I/O object. A maximum of 15 I/O objects can have `io_update_occurs` and `io_changes` events. The syntax of the event expressions all require the event type to be followed by the object name, in parentheses. For more information, see Appendix C.

***Too many include files***

A maximum of 254 files may be opened in a single compilation. The source file and the three compiler helper files count as four files altogether, thus there may be no more than 250 include files. (An include file included from another include file counts as a separate file.)

***Too many include search directories specified***

A maximum of 20 directories may be specified in the include-directory search list.



### ***Too many initializers***

A set of initializers (in braces '{' and '}') has too many members for the aggregate (array, struct, or union) being initialized.

### ***Too many msg-tags declared***

A maximum of 15 message tags can be declared per node. These can be any combination of bindable and nonbindable message tags.

### ***Too many network variables declared***

A maximum of 62 network variables can be declared per node. These can be any combination of input and output variables. Each element of an array network variable counts separately. Additional network variables can be accessed using explicit network variable updates and the fetch network management command.

### ***Too many parameters passed to function***

The compiler outputs these diagnostics when the number of actual parameters or the actual parameter types, do not match those in the prototype, and they cannot be automatically converted.

### ***Too many static declarations in this compilation***

A maximum of 32,767 static variables can be declared in a single module.

### ***Too many timers declared***

NEURON C supports a maximum of 15 application timer objects. The types of these timers do not affect the capacity.



### ***Too many when clauses***

NEURON C places entries in a table that represents the when clauses. This information is used by the NEURON CHIP firmware scheduler. The entries are variable sized, as some event expressions are more complex than others. The table size is limited to 256 bytes. When the table is full, no more when clauses can be accepted. Note that the limit is on the number of when clauses and not on the number of when tasks.

### ***Type defaults to 'int'***

The definition of ANSI C permits a declaration at file scope without a type. Likewise, functions may be declared without a return type. Such declarations must default to int, by the ANSI definition. However, such declarations are poor programming practice, and may even indicate an error, thus the compiler issues a warning diagnostic.

Consider the following example:

```
unsigned long x1, x2; x3;
```

Note the semicolon following x2. This is most likely a typo, however, ANSI C permits this and results in x3 being declared by default as an int. Due to white space rules, this appears the same to the compiler as the following declaration:

```
unsigned long x1, x2;  
x3;
```

This is almost certainly *not* what the programmer intended, yet most C compilers do not issue a warning in these circumstances.

### ***Type mismatch for binary operation***

#### ***Type mismatch in assignment expression***

These error diagnostics result from combining expressions of conflicting types, such as assigning an int to a pointer, or a pointer of one type to a pointer to another type, or in using objects that have no value (such as message tags or I/O object names) in expressions.

In many cases in ANSI C, you must use an explicit type cast. However, note that casting should be avoided if possible, as it is often poor programming practice.

#### ***Type mismatch in function parameter***

The compiler outputs these diagnostics when the number of actual parameters or the actual parameter types, do not match those in the prototype, and they cannot be automatically converted.

#### ***Type mismatch in function redeclaration***

This diagnostic indicates that a function prototype does not match a subsequent prototype, or the definition of the function. The prototypes and definition must match in terms of their storage class (for example, static, eeprom, ram) as well as their return types and their number and types of parameters.

#### ***Unacceptable function return type***

A function in NEURON C cannot return an object that is a struct or union type, or that is an array. However, a function *can* return pointers to such objects.

#### ***Unexpected END-OF-FILE in source file***

An incomplete source construct unexpectedly ended in an end-of-file condition. This may indicate mismatched brace characters { and }.

### ***Unsupported preprocessor directive 'directive'***

The following ANSI C preprocessor directives are *not* supported in NEURON C:

```
#if
#ifdef
#elif
#else
#endif
#file
#ifndef
#undef
#line
```

### ***Unterminated character constant***

The proper format of a character constant is 'value'. The 'value' can either be a single character (except ' '), or any of a number of ANSI C escape sequences.

### ***Unusual use of function address as value***

This message would occur in a situation like the following:

```
int f(void) {return 0;}

...
int g(void) {
    int x;
    x = 1;
    if (f) { // Unusual use of function address
        x = 2;
    }
    return x;
}
```

Technically, C permits such a use as shown. Such an expression has little use in NEURON C, however. The programmer most likely wanted to specify "if (f()) { ...". In other words, the rules of ANSI C allow a syntax which is most likely a programming error, and the NEURON C compiler flags it for you.

***Use of Neuron C feature is not permitted***

This message occurs when compiling a file with a .C extension. The NEURON C compiler will flag all uses of NEURON C features with this error message. Normally, a NEURON C program has a .NC extension.

***Use only 15, 10, or 1 for kbaud rate value***

Bitshift I/O object types can have their bit rates specified with either the baud or kbaud I/O declaration modifier. If kbaud is used, the only legal values are 15, 10, and 1. If baud is used, the only legal values are 15000, 10000, and 1000. The default baud rate for these I/O object types is 15 kbaud, and need not be specified.

***Use only 20, 10, or 1 for kbaud rate value***

Neurowire I/O object types can have their baud rates specified with either the baud or kbaud I/O declaration modifier. If kbaud is used, the only legal values are 20, 10 and 1. If baud is used, the only legal values are 20000, 10000, and 1000. The default bit rate for these I/O object types is 20 kbps, and need not be specified.

***Use only 4800, 2400, 1200, or 600 for I/O object's baud***

The serial I/O object type can only have a bit rate value of 600, 1200, 2400 or 4800. The bit rate value defaults to 2400 if not specified.

***Use only 15000, 10000, or 1000 for I/O object's baud***

Bitshift I/O object types can have their bit rates specified with either the `baud` or `kbaud` I/O declaration modifier. If `kbaud` is used, the only legal values are 15, 10, and 1. If `baud` is used, the only legal values are 15000, 10000, and 1000. The default baud rate for these I/O object types is 15 kbps, and need not be specified.

***Use only 20000, 10000, or 1000 for I/O object's baud***

Neurowire I/O object types can have their bit rates specified with either the `baud` or `kbaud` I/O declaration modifier. If `kbaud` is used, the only legal values are 20, 10 and 1. If `baud` is used, the only legal values are 20000, 10000, and 1000. The default bit rate for these I/O object types is 20 kbps, and need not be specified.





# Appendix F System Error Messages

This appendix lists and describes the NEURON CHIP firmware system error messages.

All of these system errors are logged by the NEURON CHIP firmware. Certain run-time errors are checked only in the development environment. These errors are marked by an asterisk (\*) in the following list.

**Already preempted. (135)**

If a program is already in preemption mode and tries to initiate another message, this error is generated. This error causes a NEURON CHIP reset.

**Application buffer too small (156)**

A message was received into network buffer but could not fit into an application buffer. May need to increase buffer size with `app_buf_in_size` pragma.

**Divide by zero error. (148)**

The application program executed a division by zero.

**Illegal send. (145) \***

This error occurs if an application program tries to send a response or a message without first building one.

**Invalid application error. (149)**

This error occurs if an application program tries to log an application error with an error out of range. The legal range is 1 to 127.

**Invalid response allocation. (137) \***

This error occurs if an application program tries to allocate (build) a response when it hasn't received a request.

**Incomplete message. (142) \***

This error occurs if an application program tries to send a message without first setting a message code or data.

**io\_in or io\_out not ready. (157) \***

`io_in()` or `io_out()` invoked for a parallel I/O object when not in proper state for input or output, respectively.

**No message available. (144) \***

This error occurs if an application program tries to reference the `msg_in` message object when no `msg_arrives` event has occurred.

**Preemption mode timeout. (134)**

The program ran out of buffers and the system gave up trying to get them. Increase the node timeout if this message occurs often. This error causes a reset.

**Read past end of message. (139) \***

This error occurs if an application program tried to read beyond the specified length of the message.

**Synchronous NV update lost.(136) \***

A synchronous network variable update was lost because the node was already in preemption mode.

**Transceiver register address out of range. (154) \***

The valid range for transceiver status information is 1 through 7.

**Write past end of application buffer. (156)**

The incoming message could not fit into the incoming application buffer.

**Write past end of message. (140) \***

This error occurs if an application program tried to write past the specified end of the message.

**Write past end of network buffer. (151) \***

The outgoing application message could not fit into the outgoing network buffer. The maximum length is 255 bytes.

*The following errors could occur if the network configuration is invalid, if the network management tool is malfunctioning or if the NEURON CHIP firmware image is corrupted. If these errors occur, try reloading the node.*

**Authentication mismatch. (160)**

A network variable message or network management message was rejected because of an authentication failure. Could be due to an authentication key mismatch or a lack of an authentication indicator in the original message. This error could indicate attempts of an intruder to "break in" to the network.

**Bad event. (129) \***

**Bad address type. (133)**

**Invalid address table index. (141)**

**Invalid domain. (138)**

**Invalid NV index. (147) \***

**Memory allocation failure. (150) \***

This error occurs if there is not enough RAM available for the functions specified in the program.

**Subnet partition detected. (159)**

Logged in a router when a subnet is detected on both sides of that router.

*The following errors may occur rarely, due to network transmission problems.*

**NV length mismatch. (130)**

The length of the data in a network variable update message is inconsistent with the length expected by the node. Rebuild and reload the images.

**NV message too short. (131)**

This error could occur if a network message is corrupted.

**NV update received for output network variable. (143)**

Another node tried to update an output network variable.

**Unknown PDU. (146) \***

This error could occur if a packet was corrupted on the network.

*The following errors may occur rarely due to NEURON CHIP, transceiver, or application failure.*

**Checksum error over application program. (152)**

**Checksum error over configuration data. (153)**

The NEURON CHIP retains a checksum of the application program and of the configuration data. If it is not the correct value, an error is logged, and the node goes into a blank or unconfigured state. This is usually a hardware problem, although it could be caused by the application writing over itself.

**EEPROM write failure. (132)**

This error occurs if too many erase/write cycles have been performed. Up to 10,000 erase/write cycles per byte can be performed in the EEPROM.

**Self test failed. (158)**

The Neuron failed its self test. The self test includes tests of RAM and internal timer and counter logic.

**Transceiver register operation timeout occurred. (155) \***

A transceiver hardware failure occurred and the transceiver could not be configured.

*The following are logged temporarily in the error log and should not be construed as errors.*

**Self-installation semaphore. (161)**

Can appear during invocation of self-installation functions.

**Read write semaphore. (162)**

Can appear during reload of an application.



# Index

3120 see *Neuron 3120 Chip*  
3150 see *NEURON 3150 CHIP*  
& operator 3-12  
/\* \*/ 1-21  
// 1-21  
@ (at-sign) D-4  
` (accent-grave) D-4

---

## A

abs built-in function B-23  
    definition and syntax C-25  
absolute value C-25  
abstract declarator, syntax B-14  
access\_address function  
    and the NEURON 3120 CHIP 6-31  
    definition, syntax and example C-25  
access\_domain function  
    and the NEURON 3120 CHIP 6-31  
    definition, syntax and example C-26  
access\_nv function  
    and the NEURON 3120 CHIP 6-31  
    definition, syntax and example C-27  
accuracy  
    low and high duration 2-61, 2-62  
    repeating timers 2-63  
    second timers 2-64, 2-65

- ACKD 4-8, 4-17, 4-24, B-8
  - for sending messages 4-22
- acknowledged messages 3-6
- acknowledged service C-78
- acknowledging a message (figure) 4-22
- addr\_table\_index built-in function
  - definition, syntax and example C-27
- address table 6-2
- addressing modes 3-16
- allocate response object C-62
- allocating buffers 4-37
- ANSI C
  - compared to Neuron C-20
  - references about vii
- app\_buf\_in\_count pragma directive 1-27, 6-10
- app\_buf\_in\_size pragma directive 1-27, 6-9
- app\_buf\_out\_count pragma directive 1-27, 6-7
- app\_buf\_out\_priority\_count pragma directive 1-27, 6-7
- app\_buf\_out\_size pragma directive 1-27, 6-7
- application buffers 6-3 through 6-11
  - application and network buffers (figure) 6-3
  - buffer size 6-5
  - outgoing
- application
  - codes
    - ranges 4-10, 4-11
    - design steps 1-9
  - errors, logging 5-16
  - image 1-3
  - messages 4-11
  - off line C-36
  - program restart C-28
  - programming 1-3
  - programming interface (API) 1-5
  - restarting 5-16
- application\_restart function 5-16
  - definition, syntax and example C-28
- array size D-9
- asynchronous and direct event processing 4-28
- auth bind-info option B-8
- authenticated bind-info option B-8
- authenticated (auth) keyword 3-42, 4-8, 4-8, C-78, C-79

- authenticated variables and messages, declaring 3-42
- authentication 3-41
  - and buffer use 6-6
  - and system response time 3-41
  - how it works 3-43, 3-44
  - key, specifying 3-43
  - process (figure) 3-44
  - using 3-42, 4-43
- auto storage class 1-23
- auto variable 1-24, B-9
- automobile application, example of connecting multiple nodes 3-17

---

## B

- backspace D-4
- baud io option B-5, C-129
- bcd2bin built-in function B-23
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-29
- behavior of writer and reader nodes 3-6
- BIF file C-78
- bin2bcd built-in function B-23
  - definition, syntax and example C-30
- binary constants 1-21
- bind bind-info option B-8
- bind-info 3-42, B-7
- binder C-79
  - program 3-5
  - network addressing 3-16
- binding information, in network variable declarations C-78
- bit direct I/O object 2-17, B-4, C-40, C-42
  - description, syntax and usage C-96
  - examples C-97
  - reverse function C-69
- bit output, used for chip select 2-59
- bit-field allocation D-11
- binary coded decimal to binary number C-29
- binary number to binary coded decimal C-30
- bind-info see *connection info*

- bitshift direct I/O object 2-17, B-4, C-40, C-42
  - description and syntax C-98
  - input example C-99
  - output example C-100
  - usage C-99
- bitwise operations D-8
- block transfers of data 4-12, 4-13
- boolean 1-22
- buffer allocation
  - compiler directives for 6-6
  - incoming application 6-9
  - incoming network 6-9
  - outgoing applications 6-7
  - outgoing network 6-8
- buffer counts 6-6
- buffer size 6-4
  - application 6-5
  - errors 6-5
  - network 6-5
  - values for (table) 6-11 through 6-13
- buffers 4-26, 4-27, 4-36
  - allocating 4-27, 4-27, 6-3
  - allocating explicitly 4-37
  - application 6-3
  - application, incoming 6-9
  - application, outgoing 6-7
  - application, size of 6-5
  - network 6-3
  - network, size 6-5
  - network, incoming 6-9
  - network, outgoing 6-8
  - size 6-4
- built-in functions C-23, C-24
- bypass mode 5-6
  - going offline in 5-9
- byte direct I/O object 2-17, B-4, C-40, C-42
  - description and syntax C-100
  - examples C-101
  - usage C-101

## C

- cancel message function C-51
  - definition, syntax and example C-51
- cancel response function C-63
  - definition, syntax and example C-63
- carriage return D-4
- certification and using SNVTs 1-7
- chip select 2-51
  - pulse 2-42
  - pin C-106
- clear\_status function 5-17
  - definition, syntax and example C-31
- clock input range and resolution, table of C-114
- clock io option B-5, C-102, C-111, C-113, C-119, C-122, C-124, C-133
- clockedge io option B-5, C-98, C-133
- clocks
  - calculating accuracy with a full-speed clock 2-62
  - calculating accuracy with other clock speeds 2-62
- code keyword 4-7, 4-8, 4-17
- command-driven vs. data-driven protocols 1-8
- COMM\_IGNORE option 5-12 through 5-14
- comment style 1-21
- comparison of resp\_arrives and msg\_succeeds 4-36
- compiler
  - directives 1-27
    - for buffer allocation 6-6
  - error messages E-3
- completion events
  - unqualified 4-25
    - processing
      - asynchronous 4-28
      - direct 4-28
      - for messages 4-24
    - success/failure (table) 4-24
  - testing
    - comprehensive 3-33, 3-34
    - partial 3-33
- completion events for messages
  - processing 4-24, 4-25
- completion events for network variables 3-24
  - processing 3-33

- comprehensive completion event testing 3-33, 3-34
- conditional events
  - syntax B-16, B-17
- config
  - keyword 6-17, B-8, B-9, C-78, C-79
  - storage class 1-23, 3-9, C-77
- config, nonconfig keywords in authentication 3-42
- config\_data built-in variable C-83
- connecting multiple nodes, example of 3-17
- connecting network variables 3-16
- connection-info 3-10, C-78
  - bind-info 3-42, C-78
- const
  - storage class 1-23, 3-9, B-9, C-77
  - variables 1-24
- constants
  - binary 1-25
  - built-in, syntax B-22, B-23
  - hexadecimal 1-25
  - integer 1-25
  - octal 1-25
- constructing a message 4-6
- CONTROL.H include file C-22
- critical section boundary and the post\_events function C-60
- critical section 3-7, 4-16, C-52
- Ctrl-Z D-5

---

## D

- data, block transfers of 4-12
- data declaration, syntax B-2
- data-driven vs. command-driven protocols 1-8
- data field, variable size 4-2
- data keyword 4-8, 4-17
- data word shift C-98
- DATE and TIME macros D-14
- declaration specifiers, syntax B-7
- declarations D-2
- declarator syntax B-13
- declarators, number of D-12



## declaring

I/O objects 2-21

guidelines 2-22, 2-23

network variables 3-4, 3-7, 3-8

Synchronous Network Variables 3-31

timers 2-12

ded io option B-5, C-113, C-119, C-121, C-131

default when clause, importance of 4-18

delay function 2-66

definition and example C-32

example C-33

delay, scaled

definition and syntax C-69

example C-70

dest\_addr 4-9

diagnostic keywords D-2

diagnostic status, return C-66

diagnostics

lexical E-2

preprocessor E-2

semantical E-2

syntactical E-2

diagram of standard program identification fields (figure ) 1-35

direct event processing 2-6, 2-7, 4-28

direct I/O objects 2-17, 2-34

disable\_mult\_module\_init pragma directive 1-28

disable\_servpin\_pullup pragma directive 1-28

disable\_snvt\_si 1-28

domain table 6-3

Duration of timeout events figure 2-64

## E

- ECHELON.H include file 1-24, C-22
- EECODE 1-24, 6-15 through 6-18
- EEFAR 1-24, 6-15 through 6-17, 6-19, 6-21
- EENEAR 1-24, 6-15 through 6-21
- EEPROM 1-24, 6-4 through 6-30
  - erase/write cycle 1-24, 3-9
  - fitting on a NEURON 3120 CHIP 6-24 through 30
  - memory map figure 6-15
  - memory region and its areas 6-14 through 6-17
  - on-chip, address table 6-2
  - on-chip, domain table 6-3
  - on-chip, relocating, 6-2
  - storage class 1-24
  - usage tip 6-23, 6-24
  - use 6-23
  - write timer 2-65
- eprom keyword 1-24, 6-20, B-9
- eprom storage class 3-9, C-77
- enable\_io\_pullups pragma directive 1-28
- enable\_multiple\_baud pragma directive 1-29, C-98, C-106, C-129
- enable\_sd\_nv\_names pragma directive 1-29
- end-of-file marker D-5
- end-of-line D-4, D-5
- enum-type syntax B-11
- enum variable type B-11
  - predefined 1-22
- enumeration syntax B-11
- EOT D-5
- error
  - handling 5-15
  - error log, size 5-16
  - messages, system F-2
  - number
    - write to log C-33
  - status, access 5-17
  - system 5-17
- error\_log function 5-16
  - definition, syntax and example C-33
- escape sequences D-6

- event
  - definition 2-4
  - reset 2-8
  - timer\_expires 2-14
  - types used in when clauses 2-5
- event scheduler 2-2
- event-driven vs. polled scheduling 1-8
- events
  - completion 3-24, 3-33
  - conditional B-16
  - directory of C-4
  - network variables and 3-24
  - predefined 2-5, 2-6, C-2
  - user-defined 2-5, 2-6, 2-8
- expected, low, and high duration of timeout events (figure) 2-64
- explicit messages 4-2, 4-5
- explicit messages vs. network variables 4-2
- explicit\_addressing\_off pragma directive 1-29
- explicit\_addressing\_on pragma directive 1-29
- expression syntax B-19, B-20
- extern storage class 1-23, B-8
- external definitions, syntax B-2, B-3
- External Interface Definitions (Figure 1-3) 1-13
- external interfaces, defining 1-12

## F

- far keyword 1-24, 6-19, B-9
- fixed timers 2-60
- floating point computation 1-20, D-9
- Flow Diagram for Timer/Counter Circuits (Figure 2-1) 2/36
- flush function 5-11
  - definition, syntax and example C-34
- flush\_cancel function 5-11
  - definition, syntax and example C-35
- flush\_completes event 5-12, B-16, C-34
  - definition, syntax and example C-4
- flush\_wait event and preemption mode 4-26, 4-27
- flush\_wait function 4-27, 4-28, 4-29
  - and the NEURON 3120 CHIP 6-31
  - definition and syntax C-35
  - example C-36

- flushing the NEURON CHIP 5-11
- for condition, built-in B-19
- foreign frames 4-11
- free msg\_in object C-52
- free resp\_in object C-64
- frequency timer/counter I/O object B-4, C-42
  - description and syntax C-102
  - example C-103
  - period range table C-102
  - usage C-103
- function declarators B-15
- function prototype feature, use of 1-21
- function prototypes 2-11
- functions
  - built-in B-22
  - directory of C-22
  - relationship between I/O measurements, objects and 2-34
  - table of C-24

---

## G, H

- global data 1-22
- go\_offline function
  - definition and syntax C-36
  - example C-37
- go\_unconfigured function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-37
- going offline in the bypass mode 5-9
- handshake
  - flag 2-48
  - line 2-44, 2-45
  - protocol 2-46
- handshake protocol sequence between master and slave (figure) 2-46
- hard pin direction I/O object 2-25

---

/

- identifiers D-3
- Identifying the Nodes (Figure 1-1) 1-11
- if condition, built-in B-19
- implementation limits, list of B-24
- implementation, defined D-2
- implicit messages 4-4
- include directive D-13
- incoming messages, format of 4-16
- initial\_value 3-10
- initializer\_list 3-10
- input
  - bit C-96
  - bitshift C-98
  - byte C-100
  - leveldetect C-104
  - neurowire C-105
  - nibble C-109
  - ontime C-113
  - parallel C-115
  - period C-119
  - pulsecount C-121
  - quadrature C-127
  - serial C-129
  - totalcount C-131
- input clock frequency 2-60
- input data, read C-39
- input edges, number of C-121, C-131
- input keyword 3-5
- input object type vs. returned data type C-40
- input objects
  - bitshift, io\_in syntax C-40
  - neurowire, io\_in syntax C-40
  - parallel, io\_in syntax C-41
  - serial, io\_in syntax C-41
- input signal, high period C-113
- input value, request latest C-58
- input\_is\_new built-in variable B-22, C-84
  - definition 2-29, C-84
- input\_value built-in variable B-22, C-84
  - definition 2-32, C-84

integer type ranges D-1

interactive devices D-3

invert io option B-5, C-102, C-111, C-113, C-119, C-121, C-122,  
C-124, C-131, C-133, C-137

I/O devices 1-6

multiple 1-11

pin assignment table 2-24

table of 2-24

I/O events 2-30

I/O functions

for timer/counter objects 2-36

performing 2-26

I/O measurements, relationship between outputs, functions and 2-34

I/O multiplexing 2-36

I/O object declaration, syntax B-2

I/O object 1-2, 1-14, 2-2, 2-17

declaring 2-16, 2-21

direct 2-17, 2-34

directory C-95

initializing C-38

optional definitions C-39

overlying 2-25

parallel 2-18, 2-34

serial 2-18, 2-34

syntax C-95

table of types 2-19

timer/counter 2-17, 2-35

types 2-17

names C-39, C-40

I/O object types

bit C-96, C-104

bitshift C-98

byte C-100

neurowire C-105

nibble C-109

parallel 2-18, 2-26, 2-34, C-115

serial 2-18, 2-26, 2-34, C-129

I/O option declarations, syntax B-4

I/O pin 2-21



I/O resources, use of 2-22  
`io_types.h` include file C-39  
`io_change_init` built-in function 2-16, 2-26, B-23, C-6  
    definition, syntax and example C-38  
`io_change_event` B-17  
`io_changes` event 2-16, 2-30, B-17, C-84  
    definition and syntax C-4  
`io_in` built-in function 2-16, 2-26, 2-27, 2-32, B-23, C-84  
    definition and syntax C-39  
    example C-41  
    with when clauses 2-34  
`io_in_ready` event B-23, C-116  
    definition and syntax C-7  
`io_out` built-in function 2-16, 2-26, 2-28, B-23  
    definition and syntax C-41  
    example C-43  
`io_out_ready` event B-23, C-116  
    definition and syntax C-8  
`io_out_request` built-in function 2-16, 2-26, B-23, C-116  
    definition and syntax C-43  
`io_select` built-in function 2-16, 2-26, 2-36, 2-37, B-23  
    definition, syntax and example C-44  
`io_set_clock` built-in function 2-16, 2-26, 2-37, B-23  
    definition, syntax and example C-45  
`io_set_direction` built-in function  
    definition, syntax and example C-46  
`io_update_occurs` event 2-16, 2-31, B-17, C-84  
    definition and syntax C-9  
`is_bound` built-in function 3-16, B-23  
    definition, syntax and example C-47

---

## K

`kbaud` io option B-5, C-98, C-106

---

## L

`lamp` program 4-19  
`lamp/switch` example 1-11  
    polled version of 3-37  
    with explicit messages 4-19  
`larger` value 4-16

- leveldetect direct I/O object 2-17, B-4, C-40
  - description and syntax C-104
  - usage C-104
  - example C-104
- lexical diagnostics E-2
- library functions 1-21
- Limits.h include file B-24, D-7
- linker 6-17, 6-18
- logging system errors 5-17
- LONBUILDER Developer's Workbench 1-4
- LONBUILDER User's Guide viii, 1-4
- long int 1-20, D-7
- long io option B-5, C-124
- long to short integer conversion D-7
- LONMANAGER API Programmer's Guide 1-3, 1-4
- LONTALK messages 1-7

## M

- main
  - use of D-3
- master 2-42 through 48, C-115, C-119
  - master/slave A 2-50
  - master/slave B 2-51
- master io option B-5, C-106
- master\_read state 2-47
- master/slave A 2-50
  - transferring data from master to slave 2-50
  - transferring data from slave to master 2-50, 2-51
- master/slave B 2-51
  - transferring data from microprocessor to slave B figure 2-51
- master\_write state 2-47
- max built-in function B-23
  - definition, syntax and example C-48
- max\_rate\_est option 4-10, B-8, C-78, C-80, C-87
- Media Access (MAC) layer 4-3
- memcpy built-in function 1-21, 4-13, B-23
  - definition, syntax and example C-49
- memset built-in function 1-21, B-23
  - definition, syntax and example C-49

- memory 6-14 through 6-30
  - areas 6-16
  - default 6-18
  - EEPROM use 6-23
  - fitting on a NEURON 3120 CHIP 6-24 through 6-30
  - non-default and special keywords for 6-19
  - off-chip 6-14
  - RAM use 6-22
  - regions 6-15
  - using NEURON CHIP 6-14
- memory mapped I/O 6-14
  - figure summarizing 6-15
  - usage tip for 6-23, 6-24
- messages 1-7
  - acknowledging (figure) 4-22
  - cancelling 4-14
  - codes 4-6, 4-10
    - ranges 4-11, C-4
  - completion status 4-22, 4-23
  - constructing 4-6
  - data block transfer 4-12, 4-13
  - error diagnostics E-2
  - explicit 1-8
  - FATAL diagnostics E-2
  - format of incoming 4-16
  - FYI (for your information) E-2
  - incoming, format of 4-16, 4-17
  - preemption mode and 4-26
  - processing completion events for 4-24 through 4-26
  - rate
    - maximum C-80
    - of network variables C-80
  - receiving 4-15
  - sending 4-14
    - (figure) 4-3
  - status, monitor C-34

- messages (cont)
  - tag declaration 4-6, 4-9, 4-10
    - definition, syntax and example C-87
  - tags
    - connected C-47
    - connecting 4-20
    - declaring 4-9
    - non-bindable 4-21
  - unwanted 4-18
  - warning E-2
- micro\_interface pragma directive 1-29
- Microprocessor Interface Programmer's Guide 3-5
- millisecond timer 2-12
- min built-in function B-23
  - definition, syntax and example C-50
- Miranda prototype rule 2-11
- modifiers, network variable 3-8
- monitoring network variables 3-39
- msg\_alloc built-in function 4-37, B-23
  - definition, syntax and example C-50
- msg\_alloc\_priority built-in function 4-37, B-23
  - definition, syntax and example C-51
- msg\_arrives event 4-15, B-17
  - definition, syntax and example C-10
- msg\_cancel built-in function 4-14, B-23
  - definition, syntax and example C-51
- msg\_completes event 4-22, 4-27, B-17
  - definition and syntax C-10
  - example C-11
- msg\_fails event 4-22, B-17
  - definition, syntax and example C-11
- msg\_free built-in function 4-37, B-23
  - definition, syntax and example C-52
- msg\_in built-in object 4-16, 4-17, B-22, C-87
- msg\_out built-in object 4-6, B-22, C-88
  - allocating buffers and 4-37
  - definition, syntax and example 4-6, C-87
- msg\_receive built-in function 4-16, 4-30, 5-7, B-23
  - definition and syntax C-52
  - example C-53
- msg\_send built-in function 4-6, 4-14, 4-27, B-23
  - definition, syntax and example C-53

- msg\_succeeds event 4-22, B-17
  - comparison with resp\_arrives 4-36
  - definition, syntax and example C-12
- msg\_tag 4-6, 4-9, B-7
- mtimer 2-12, 2-60, B-10, C-82
  - accuracy 2-61
  - and clock speed 2-62
- muldiv function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-54
- muldivs function
  - and the NEURON 3120 CHIP 6-31
  - definition and syntax C-54
  - example C-55
- multi-character displays example 2-59
- multibyte characters D-6
- multicast connections and buffer use 6-6
- multiple I/O devices connected to a single node (figure ) 1-11
- multiplexed timer/counter circuit
  - ontime input object and C-113, C-114
  - period input object and C-119, C-120
  - pulsecount input object and C-121, C-122
  - totalcount input object and C-131, C-132
- mux io option B-5, C-113, C-119, C-121, C-131

## N

- names, unacceptable C-93, C-94
- net\_buf\_in\_count pragma directive 1-29, 6-9
- net\_buf\_in\_size pragma directive 1-30, 6-9
- net\_buf\_out\_count pragma directive 1-30, 6-8
- net\_buf\_out\_priority\_count pragma directive 1-30, 6-8
- net\_buf\_out\_size pragma directive 1-30, 6-8
- netvar\_processing\_off pragma directive 1-30
- netvar\_processing\_on pragma directive 1-30
- network buffers 6-3
  - buffer counts 6-6
  - buffer size 6-4, 6-5
  - incoming 6-9
  - outgoing 6-8
  - tables of count and size values 6-11
- network diagnostic messages 4-11
- network image 1-6



- network keyword 3-5
- network management messages 4-11
- network management node 1-3 through 1-6, 3-12, 5-8, C-79, C-80
  - and address table 6-2
- network storage class 1-23, 1-24
- network variables 1-2, 1-14, 4-11
  - advantages of 1-6, 1-7
  - and arrays 3-12
  - and performance 3-31, 3-32
  - authentication of C-79
  - benefits 1-8
  - classes 3-9, C-77
  - connected C-47
  - connecting 3-16
    - example of 3-17 through 3-19
  - connecting writers to readers 3-3, 3-4
  - connection information 3-10, 3-42
  - declaring 3-4 through 3-13, C-76
    - connection information (connection-info) C-78 through C-81
    - examples of 3-14
    - network variable classes (class) C-77
    - network variable modifiers (netvar-modifier) C-76
    - network variable types (type) C-77
    - synchronous 3-31
  - declaring as polled 3-37
- events 3-24
  - nv\_update\_completes event description and example 3-27
  - nv\_update\_fails event description and examples 3-26
  - nv\_update\_occurs event description and examples 3-24, 3-25
  - nv\_update\_succeeds event description and examples 3-26
  - sample program 3-28 through 3-30
- example of declarations 3-14
- explicit messages vs. 4-2
- implicit messages and 4-4
- initializers for 3-10
- input/output 1-14, 1-15
- keywords B-10
- message rate of C-80
- modifiers 3-8, C-76
- monitoring 3-39
- net-var type B-10



## network variables (cont)

### polling of 3-34, 3-35

- declaring a network variable as polled 3-37, 3-38

- example of 3-36

### processing completion events 3-33

### readers and writers of 1-14, 1-15

### service type used for C-77

### size 3-11, 3-12

### standard network variable types (SNVTs) 1-7, 3-12, 3-13

### synchronous 3-30, 3-31, 3-32

### synchronous vs. nonsynchronous 3-31, 3-32

### syntax 3-5 through 3-7, C-76

### types 3-11, 3-12, C-77

### update order 3-30, 3-31

### updates 3-7

## NEURON 3120 CHIP

- fitting a program on 6-24 through 6-30

- system library 6-31

## NEURON 3150 CHIP off-chip memory 6-14 through 6-18

## NEURON C

- character set D-4 through D-6

- declarations 1-26

- external definitions B-2, B-3

- properties 1-2

- resources C-116

- storage classes 1-22 through 1-24

- syntax conventions B-2

- unacceptable names C-93, C-94

- variable classes B-9

- variable types 1-22

## NEURON C and ANSI C, differences 1-20 through 1-22

## NEURON CHIP

- 3120, memory map 6-15

- 3150, memory map 6-15

- flushing 5-11

- forced sleep 5-14, 5-15

- I/O pins 2-16

- memory 1-22 through 1-24, 6-14 through 6-30

- pins 1-14

- powering down with the sleep function

- description of function and example C-71

- syntax C-70

## NEURON CHIP (cont)

- putting to sleep 5-12 through 5-14
- reset event
  - description and syntax C-18
  - example C-19
- roles of 1-3, 1-4
- sleep mode 5-10
- wake up 5-10
- NEURON CHIP-to-NEURON CHIP interface 2-48
- NEURON emulator and application errors 5-16
- neurowire objects
  - bitshift, io\_out syntax C-42
- neurowire select
  - syntax C-105
- neurowire serial I/O object 2-18, B-4, C-40, C-42
  - description C-105, C-106
  - example C-108
  - syntax C-106, C-107
  - usage C-108
- nibble direct I/O object 2-17, B-4, C-5, C-40, C-42
  - description and syntax C-109
  - examples C-110
  - usage C-109
- node 1-3, 1-4
  - forced sleep 5-14, 5-15
  - ID 1-8
  - identification 1-5, 1-6
  - identification string 1-33 through 1-35
  - installation 1-5, 1-6
  - network management 1-3, 1-4
  - reset function (node\_reset) 4-26, 4-27, 5-8
    - description, syntax and example C-55
  - resetting 5-15, 5-16
  - unconfigure (go\_unconfigure function) C-37
- node\_reset function 5-15, 5-16
  - definition, syntax and example C-55
- nodes, certified
  - pragma 1-34
- nonauthenticated keyword 3-42, C-79
- non-bindable 4-21
- nonbind bind-info option B-8
- nonconfig keyword B-8, C-78, C-79, C-80

- nonpriority C-78
- nonpriority bind-info option B-8
- NULL pointer constant D-14
- num\_addr\_table\_entries pragma directive 1-30, 6-2
- num\_domain\_entries pragma directive 1-31, 6-3
- numbits io option B-5, C-98
- nv\_array\_index built-in variable C-58, C-85
- nv\_in\_addr built-in variable 3-40, C-85
- nv\_table\_index built-in function
  - definition, syntax and example C-56
- nv\_update\_completes event 3-24, 3-27, B-17
  - definition, syntax and example C-13
- nv\_update\_fails event 3-24, 3-26, 3-33, B-17
  - definition, syntax and example C-14
- nv\_update\_occurs event 3-24, 3-25, 3-32, 3-35, 3-40, B-17
  - definition, syntax and example C-15
- nv\_update\_succeeds event 3-16, 3-24, 3-26, 3-33, B-17
  - definition, syntax and example C-16

## O

- object database 3-16
- objects 1-2
  - built-in C-83
  - msg\_in C-87
  - msg\_out C-87
  - resp\_in C-88
  - resp\_out C-89
- offline bind-info option B-8
- offline event 5-8, B-16, C-78
  - definition, syntax and example C-17
- offline, going in bypass mode 5-9
- offline\_confirm function 5-9
  - definition, syntax and example C-57
- one\_domain pragma directive 1-31, 6-3
- oneshot duration, table of C-111
- oneshot timer/counter I/O object 2-17, B-4, C-40
  - description and syntax C-111
  - example C-112
  - oneshot duration table C-111
  - usage C-112
- online event 5-8, B-16
  - definition, syntax and example C-18

- ontime timer/counter I/O object 2-17, B-4, C-3, C-40
  - description C-113
  - example C-114
  - syntax C-113, C-114
  - usage C-114
- outgoing message, defined C-8
- output
  - alphabetical list of I/O objects C-95
  - bit C-96
  - bitshift C-98
  - byte C-100
  - frequency C-102
  - neurowire C-105
  - nibble C-109
  - oneshot C-111
  - parallel C-115
  - pulsecount C-122
  - pulsewidth C-124
  - serial C-129
  - triac C-133
  - triggeredcount C-137
- output buffer allocation
  - non-priority C-50
  - priority C-51
- output keyword 3-5
- output object type vs. returned data type C-42
- output objects 2-35, 2-36
  - bitshift, io\_out syntax C-42
  - neurowire, io\_out syntax C-42
  - parallel, io\_out syntax C-43
  - serial, io\_out syntax C-43
- output pulse delay C-133
  - table of C-133
- outputs, relationship between I/O measurements, functions and 2-34
- output\_value 2-35, 2-36
- overlying I/O objects 2-25

## P

- padding of structures D-10
- parallel interface C-115
- parallel I/O object
  - data type and C-40, C-42
  - data transfer table 2-47, 2-48
  - description C-115
  - examples 2-52, C-118
    - thermostat interface example 2-52
    - simple light dimmer interface example 2-57
    - seven-segment LED display interface example 2-59
    - neurowire connection to a display figure 2-59
  - handshake protocol 2-46
    - handshake protocol sequence figure 2-46
  - hardware interface 2-44
  - hardware interface 2-44
  - io\_out syntax C-43
  - master mode 2-42
  - modes of operation 2-41
  - NEURON C resources C-116
  - NEURON CHIP-to-NEURON CHIP interface 2-48
  - object type 2-18, B-4
    - pin table 2-19, 2-20
  - object 2-18, 2-34, 2-41
  - pin assignments figure 2-44
  - Possible Master/Slave Connections 2-43
  - slave A mode 2-42
  - slave B mode 2-42
  - syntax C-117
  - transferring data from master to slave 2-50
  - usage C-118
- parallel\_io\_interface structure C-116
- partial completion event testing 3-33
- performing I/O: functions and events 2-26
- period timer/counter I/O object 2-17, B-4, C-5, C-40, C-119
  - description and syntax C-119
  - example C-120
  - range and resolution, table of C-120
  - usage C-120
- period, total, of an input signal C-119

- pin
  - logical state control C-96
  - update event C-9
- pin assignments for the three modes of parallel I/O (figure) 2-44
- pointers 1-21, B-13, D-9
- poll 4-29
- poll built-in function 3-34, 3-35, B-23
  - definition and syntax C-58
  - example C-59
- polled
  - applications 1-8
  - declaring a network variable as 3-37
  - keyword B-10
  - network variable modifier 3-8, C-76
  - polled scheduling vs. event-driven 1-8
- polling
  - and multiple updates C-58
  - definition of 3-34, 3-35
  - network variables 3-34
- possible master/slave connections for the NEURON CHIP (figure) 2-43
- post\_events function 5-6
  - definition, syntax and example C-60
- post\_events, msg\_receive function 5-7
- power\_up function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-61
- pragma directives 1-27
  - list and definitions of 1-27 through 1-34
- predefined events 2-5, 2-6, C-2, C-3
  - I/O-related 2-30
  - limited to once/program C-3
  - list of 2-6
- preemption mode 3-31, 3-32, 4-26, 6-6
- preprocessor
  - diagnostics E-2
  - directives 1-21, 1-22, 1-27 through 1-34, D-13, D-14
- priority C-78
  - bind-info option B-8
  - definition 2-4
  - when clauses 2-10, B-16
- priority\_on keyword 4-6
- program identification fields 1-34, 1-35



programming model 1-3

protocols

data-driven vs. command driven 1-8

prototypes, function 2-11

pullups, enabling 1-28

pulse generator C-122

pulsecount timer/counter I/O object (input) 2-17, B-4, C-3, C-40

description, syntax and usage C-121

example C-122

pulsecount timer/counter I/O object (output) 2-17, C-42, C-122

description and syntax C-122

example C-123

pulse period table C-123

usage C-123

pulsewidth timer/counter I/O object 2-17, B-4, C-42

8-bit pulsedwidth output control range table C-125

16-bit pulsedwidth output control range table C-125

description and syntax C-124, C-125

usage and example C-126

---

## Q

quadrature timer/counter I/O object 2-17, B-4, C-5, C-40

description and syntax C-127

usage and example C-128

---

## R

RAM 6-14 through 6-16

fitting on a NEURON 3120 CHIP 6-26

usage tip for memory mapped I/O 6-23, 6-24

use 6-22

ram keyword 1-24, B-9

for functions 6-20

RAMCODE 6-15, 6-20

RAMFAR 6-15, 6-19

RAMNEAR 6-18 through 6-20

ram\_test\_off pragma directive 1-31

random function

definition, syntax and example C-61

random number function C-61

rate\_est option 4-10, B-8, C-78, C-87

read\_only\_data built-in variable C-86

- read\_write\_protect pragma directive 1-32
- reader nodes 3-6, 3-37, 3-38
- receive message function C-52
- receive response function C-65
- receive transactions, number of 6-10
- receive\_trans\_count pragma directive 1-32, 6-10
- receiving a message 4-15
- reference value, for io\_changes C-5
- refresh\_memory function
  - definition, syntax and example C-62
- register class 1-20, B-9
- registers D-10
- relinking the program 6-21
- remainder D-8
- repeating timers 2-12, 2-63, B-10, C-82
- REQUEST 4-9, 4-17, 4-23, 4-24, 4-29
- request/response mechanism
  - for messages 4-29
  - with explicit messages 4-2
- request/response mechanism, using 4-29
- reserved words 1-21
  - list of C-90 through C-93
- reset command 5-8
- reset event 2-8, B-16
  - definition, syntax and example C-18
- reset node C-55
- reset pin 5-15, 5-16
- resetting the node 5-15, 5-16
  - disadvantages 5-2
- resp\_alloc built-in function 4-38, B-23
  - definition, syntax and example C-62
- resp\_arrives event 4-32, B-17
  - comparison with msg\_succeeds 4-36
  - definition, syntax and example C-19
- resp\_cancel built-in function, B-23
  - definition, syntax and example C-63
- resp\_free built-in function 4-38, B-23
  - definition, syntax and example C-64
- resp\_in built-in object. 4-33, B-22, C-88
- resp\_in\_addr 4-34
- resp\_out built-in object 4-12, 4-31, B-22, C-89

- resp\_receive built-in function 4-32, 4-33, 5-7, B-23
  - definition, syntax and example C-65
- resp\_send built-in function 4-32, B-23
  - definition, syntax and example C-66
- response 4-31 through 4-36
  - comparison of resp\_arrives and msg\_succeeds 4-36
  - constructing 4-31
  - examples 4-34
  - format 4-33
  - incoming, structure C-88
  - outgoing, structure C-89
  - receiving 4-32
  - sending 4-32
- restarting the application 5-16
- retrieve\_status function 5-17, C-31
  - and the NEURON 3120 CHIP 6-31
  - definition and syntax C-66
  - example 5-18
- reverse built-in function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-69
- ROM
  - default memory usage 6-18
  - eeprom keyword 6-20
  - memory areas of 6-16
  - memory region 6-15, 6-16
  - off-chip 6-15, 6-16
  - usage tip for memory mapped I/O 6-23, 6-25
- routers 2-41
- RS232 communications 2-16

## S

- sample application A-2, A-3
- Sample Development Network with Five Nodes figure 1-19, 3-4
- scaled timers and I/O objects 2-61
- scaled\_delay built-in function 2-66
  - definition and syntax C-69
  - example C-70
- scheduler 2-2, 2-9, 5-2, 5-3
  - bypass mode 5-6
  - example 5-6
  - reset off 5-4

- scheduler reset mechanism 5-4
  - example 5-6
- scheduler\_reset pragma directive 1-32, 2-10, 3-27, 4-18, 5-5
- scheduling
  - event-driven vs. polled 1-8
  - of updates 3-7
  - of when clauses 2-9
- SD 1-7, 3-13
- sd\_string C-76
- second timers 2-12
  - accuracy of 2-64
- select io option B-5, C-106
- Self Documentation (SD) 1-7, 3-13
- Self-Identification (SI)
- semantic diagnostics E-2
- send message function (msg\_send) C-53
- send response function (resp\_send) C-66
- sending a Message 4-14
  - figure 4-3
  - using the ACKD service 4-2 through 4-24
- Sending a response (figure) 4-30
- serial
  - data transfer C-129
  - displays 2-59
  - I/O objects 2-18, 2-34
  - transfer object C-105
- serial I/O object 2-18, B-4, C-40, C-42
  - description and syntax C-129
  - examples C-130
  - usage C-130
- Service Interface Control Blocks (SICBs) 6-3
- service type 4-8, 4-9, 4-17
  - used for network variables C-78, C-79
- service\_type 4-6
- set\_id\_string pragma directive 1-33
- set\_netvar\_count pragma directive 1-33
- set\_node\_sd\_string pragma directive 1-33
- set\_std\_prog\_id pragma directive 1-34
- seven-segment LED display interface example 2-59
- shaft or positional encoder input C-127
- short io option B-5, C-124
- short int 1-20, D-7

- SI 1-7, 3-13
- signed char 1-22
- signed long int 1-22
- signed short int 1-22
- simple light dimmer interface example 2-57
- slave C-119
- slave A 2-42 through 2-48, 2-50, C-115
- slave B 2-42 through 2-48, 2-51, C-115, C-119
- slave io option B-5, C-106
- slave\_b io option B-5
- slave\_read state 2-48
- slave\_write state 2-47
- sleep 5-10 through 5-15
  - built-in function B-23
    - definition and example C-71
    - syntax C-70
  - forced sleep 5-14, 5-15
  - mode 5-10
  - putting the NEURON CHIP to sleep 5-12, 5-13
- smaller value function C-48
- SNVT 1-7
  - definition of 3-12, 3-13
  - ID storage 3-12
  - SNVT\_lev\_disc 3-5
- soft pin direction I/O object 2-25
- software timers
  - see *timers*
- source files, includable D-14
- square wave output signal C-102
- Standard Network Variable Types (SNVTs) 1-7, 3-12, 3-13
- Standard Program Identification Fields (Table 1-1) 1-34
- start-up, determine cause of C-61
- statement syntax B-18
- static storage class 1-23, B-9
- status command C-66
- STATUS.H C-66
- status\_error\_log C-68
- status\_lost\_msgs C-67
- status\_model\_number C-68
- status\_node\_state C-68
- status\_rcv\_transaction\_full C-67
- status\_reset\_cause C-3

- status\_struct structure C-66
- status\_timeouts C-67
- status\_version\_number C-68
- status\_xmit\_errors C-67
- stimer 2-12, 2-60, B-10, C-82
  - accuracy 2-60
- storage classes 1-22 through 1-24
- struct B-12
- structure/union syntax B-12
- suspend processing
  - formulas for delay time C-32
- switch node
  - tasks for 1-16
- switch program 4-20
- switch statement D-12
- sync io option B-5, C-133, C-137
- sync keyword 3-31, B-10
- sync network variable modifier 3-8, C-76
- synchronized io option B-5
- synchronized keyword B-10
- synchronized network variable modifier 3-8, C-76
- synchronous network variables 3-30, 3-31, 5-6
  - declaring 3-31
  - updating 3-32
- synchronous vs. nonsynchronous network variables 3-31, 3-32
- syntactic diagnostics E-2
- syntax
  - conventions for NEURON C B-2
  - summary 1-20, 1-21
  - typographic conventions for x
- system
  - errors, logging 5-17
  - overhead 5-18, 5-19
  - storage class 1-24, B-9



---

## T

- tag keyword 4-7
- tag\_identifier option 4-10, C-87
- task 2-3 through 2-5, 3-7
  - definition 2-5
  - syntax B-16
- task declarators B-16
- tasks
  - for the switch node 1-16
  - order of execution 2-10
- thermostat interface example 2-52
- timeout C-106
- timer\_expires event 2-14
- timer objects 2-2, 2-12, 2-60
  - keywords B-10
- timer/counter
  - alternate clock assignment C-45
  - dedicated 2-36
  - I/O objects 2-17, 2-35
  - I/O functions 2-36
- timer\_expires event 2-14, B-17
  - definition, syntax and example C-20
  - unqualified C-20
- timer\_name 2-12
- timers 2-12, 5-7
  - accuracy of millisecond 2-61
  - bitshift 2-61
  - calculating accuracy for software timers 2-61
  - checking for specific 2-15
  - configurable EEPROM write timer 2-61
  - declaring 2-12
  - duration, formula for 2-62
  - expires event 2-14
  - fixed duration 2-60
  - millisecond 2-12, 2-60
  - neurowire 2-61
  - number of 2-12
  - preemption mode timeout 2-60
  - pulse count input 2-60
  - repeating 2-63, 2-64
  - scaled and I/O objects 2-61

- timers (cont)
  - second 2-12, 2-60
    - accuracy of 2-64
  - serial 2-61
  - software 2-60, 2-61
  - triac pulse 2-60
  - watchdog 2-61
  - with a full-speed clock (10MHz) 2-62
  - with other clock speeds 2-62
  - write, EEPROM 2-65
- timers\_off function
  - definition, syntax and example C-71
- totalcount timer/counter I/O object 2-17, B-4, C-5, C-40
  - description and syntax C-131
  - usage and example C-132
- Transferring Data from Microprocessor to Slave B (Figure 2-6) 2-51
- transition detect, to 0 level C-104
- TRAP n diagnostic D-3, E-2
- triac timer/counter I/O object 2-17, C-42
  - description and syntax C-133, C-134
  - examples C-135, C-136
  - pulse delay table C-134
  - usage C-134
- triggeredcount timer/counter I/O object 2-17, B-4, C-42
  - description and syntax C-137
  - usage and example C-138
- two nodes using the same network variable (figure) 1-15
- type qualifiers 1-23
- typedef B-9
- types, for network variables 3-11, 3-12, C-77
- typographic conventions in this guide x

## U

- UNACKD 4-8, 4-17, 4-23, 4-24, B-8, C-78
- UNACKD\_RPT 4-8, 4-17, 4-23, 4-24, B-8, C-78
- unackd\_rpt service C-78, C-79
- unackd service C-78, C-79
- unicast connections and buffer use 6-6
- union B-12
- unions D-10
- unsigned char 1-22
- unsigned long int 1-22

- unsigned short int 1-22
- unsigned to signed integer conversion D-7
- update\_address function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-72
- update\_domain function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-73
- update\_nv function
  - and the NEURON 3120 CHIP 6-31
  - definition, syntax and example C-74
- updates
  - scheduling of 3-7
  - when occur 3-7
- user-defined events 2-5, 2-8

---

## V

- variable classes, syntax B-9
- variable declaration syntax B-6
- variable types 1-22
- variables
  - built-in B-22, C-83
  - syntax C-22
  - initialization 1-24
  - io\_in 2-27
  - io\_out 2-28
- volatile class 1-20, B-9
- volatile-qualified type D-11

---

## W

- wake up NEURON CHIP 5-10
- watchdog timer 4-26, 5-7
  - range C-75
- watchdog\_update function 5-7
  - definition, syntax and example C-75
- waveform
  - frequencies, table of C-124
  - generator C-124
- wchar\_t D-5

- when clauses 2,2 2-3
  - default, importance of 4-18
  - priority 2-10
  - scheduling of 2-9
    - scheduling of nonpriority and priority (figure) 5-3
  - types of events used in 2-5
- when statement 1-2, 2-4
- while condition, built-in B-18
- wink event 5-8, 5-10, B-16
  - definition, syntax and example C-21
- wink command 1-6, C-21
- write token 2-46, C-115
- writer node 3-41
  - behavior of 3-6